AFRL-RI-RS-TR-2011-252

MULTISCALE ARCHITECTURES AND PARALLEL ALGORITHMS FOR VIDEO OBJECT TRACKING

UNIVERSITY OF MISSOURI

OCTOBER 2011

FINAL TECHNICAL REPORT

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■**UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

AFRL-RI-RS-TR-2011-252 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/	/s/
STANLEY LIS Work Unit Manager	PAUL ANTONIK, Technical Advisor Advanced Computing Division Information Directorate

REPORT DOCUMENTATION PAGE	*Form Approved* **OMB No. 0704-0188**

1. REPORT DATE *(DD-MM-YYYY)* October 2011	**2. REPORT TYPE** Final Technical Report	**3. DATES COVERED** *(From - To)* June 2010 – April 2011

4. TITLE AND SUBTITLE MULTISCALE ARCHITECTURES AND PARALLEL ALGORITHMS FOR VIDEO OBJECT TRACKING	**5a. CONTRACT NUMBER** FA8750-10-1-0182
	5b. GRANT NUMBER N/A
	5c. PROGRAM ELEMENT NUMBER 62788F
6. AUTHOR(S) Kannappan Palaniappan	**5d. PROJECT NUMBER** T3CS
	5e. TASK NUMBER MI
	5f. WORK UNIT NUMBER SS
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Missouri W1025 Lafferre Hall Columbia, MO 65211	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/Information Directorate Rome Research Site/RITB 525 Brooks Road Rome NY 13441	**10. SPONSOR/MONITOR'S ACRONYM(S)** AFRL/RI
	11. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-RI-RS-TR-2011-252

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Implementation and performance of an extended set of parallel multicore video processing chain of modules were investigated including Cell/B.E. flux tensor for motion detection, morphology, connected component labeling, and object statistics for blob extraction. A limitation of earlier work included unsatisfactory end-to-end performance and a need for better integration within the Net-Centric Exploitation and Tracking (N-CET) software framework. We examined software integration for Phoenix support, reducing thread overhead, video processing module interoperability, enhancing streaming performance, evaluating power requirement tradeoffs between different algorithms, assisting with tighter integration of the individual modules and gathering benchmarking data to analyze performance bottlenecks in communication pathways and computational algorithms. The video processing modules were ported to the IBM QS20/QS22 Blade architectures with 16 Synergistic Processing Elements for improved numerical computation performance, especially for the flux tensor computation.

15. SUBJECT TERMS
parallel multicore algorithms, IBM Cell Broadband Engine, video processing pipeline, moving object detection, flux tensor, binary morphology, connected component labeling

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** STANLEY LIS
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UU	63	**19b. TELEPHONE NUMBER (*Include area code*)** N/A

Contents

List of Figures

List of Tables

1. SUMMARY

Implementation and performance of an extended set of parallel multicore video processing chain of modules for the Cell Broadband Engine-based (Cell/B.E.) were investigated including flux tensor for motion detection, morphology, connected component labeling, and object statistics for blob extraction. The processing goal was to achieve real time moving object detection in 2 Mpixel, 25 Hz high-definition (HD) video imagery with a bandwidth of approximately 1.5 Gb/s. A limitation of earlier work included unsatisfactory end-to-end performance, quality of Graphics Processing Unit (GPU)-based image registration and a need for better integration within the Net-Centric Exploitation and Tracking (N-CET) software framework. We examined software integration for Phoenix support, reducing thread overhead, video processing module interoperability, enhancing streaming performance, evaluating power requirement tradeoffs between different algorithms, assisting with tighter integration of the individual modules and gathering benchmarking data to analyze performance bottlenecks in communication pathways and computational algorithms. The video processing modules were ported to the IBM QS20/QS22 Blade architectures with 16 Synergistic Processing Elements for improved numerical computation performance, especially for the flux tensor computation. A technical report and publications were produced.

A summary of specific activities includes:

1. Interoperability: Tighter integration of morphology, connected component labeling (CCL) and blob statistics to enable the code to fit onto a single Cell processor and run as a single binary image in order to minimize multiple thread creations, memory management and communication overheads and achieve interoperability requirements.

2. Performance: Enhancement of the Cell/B.E. implementation of morphology, CCL, blob statistics, and flux tensor using code profiling and optimization tools to improve overall performance towards N-CET benchmarks. Extensive performance testing on SONY Playstation-3 (PS-3) and IBM QS22 versions of the Cell/B.E. multicore processor.

3. Power Tradeoffs: Examined performance optimization options and compared two or more versions of video processing modules and parallel implementations, namely for the flux tensor, to compare tradeoffs between memory, space, interoperability, interprocess communication and power requirements, in order to achieve 25 frames/sec end-to-end sustained data flow.

4. Parameter tuning: Applied performance tuning of various modules to better characterize moving object detection, blob detection and tracking performance using field experiment data such as the Stockbridge HD video sequence. This required adding feedback between the different modules to adapt to different environmental conditions and performance parameters.

5. Registration algorithm: There was a pressing need to improve the registration algorithm currently available to N-CET and delivered to AFRL by an outside contractor. The quality of the mosaics produced by this code was unsatisfactory since the algorithm is limited to translation only motion which does not adequately model the actual camera motion and perspective transformations between adjacent images captured by the moving camera. A structure tensor-based algorithm was tested on sample data provided by AFRL and evaluated for N-CET needs. This evaluation is ongoing and the sequential code is available to AFRL. Further improvement and parallelization of the structure tensor algorithm is proposed as a new activity depending on the target multicore architecture.

Further improvements were made to the flux tensor implementation for moving object detection as this is the most computationally expensive part of the video processing chain. The details of the new implementation that reorders the spatial and temporal derivatives are provided along with benchmarking data particularly related to power efficiency of the flux tensor computation. The flux tensor operator estimates significant orientations in multidimensional gridded datasets and is an efficient technique for moving object detection in video datasets. The original parallel algorithm implemented for the Cell/B.E. PS-3 architecture with six SPE computational cores achieved a speed-up improvement factor of 40 compared to the sequential

algorithm using the smallest filter sizes which offers substantial energy efficiency optimization choices. The super-linear speed-up behavior is due to the extensive use of vectorized floating point operations, fused short vector multiply add (FMA) instructions and double buffering to overlap computation and communication. Using larger filter sizes the speed-up gain decreased to a factor of 12 since the total volume of computations increases faster on the Cell/B.E. as the parallel implementation must recompute the intermediate spatial derivatives in comparison to the sequential implementation which uses significantly more memory. For larger filter sizes the limited local store on the SPEs also leads to smaller data partitioning sizes that requires more than six threads of execution which results in two stages of computation thus reducing speed-up. For all filter sizes tested the parallel flux tensor algorithm was able to exceed realtime performance requirements using a single PS-3 Cell/B.E. processor for standard definition (SD) sized video streams and for most of the filter sizes for HD sized video streams. The lower power requirements for the multicore PS-3 Cell/B.E. compared to an Intel Xeon processor makes the energy efficiency performance to power ratio of the flux tensor more than 160 times better for the smaller filter sizes and more than 50 times better for the larger filter sizes for HD video streams. The dependency of the energy efficiency factor on the flux tensor filter size provides an additional optimization dimension based on output image quality, that is using slightly smaller filter sizes for a marginal reduction in performance.

Specific deliverables for the project included:

1. Improved parallel Cell/B.E. implementation with source code of the flux tensor and thresholding modules with faster performance.
2. Improved parallel Cell/B.E. implementations with source code the blob extraction modules including morphology, CCL and blob statistics with better integration and interoperability.
3. Performance benchmarks of the improved flux tensor, thresholding, morphology, CCL and blob statistics modules using both real and synthetic datasets.
4. Energy efficiency of the multicore implementation of the flux tensor computation on the Cell/B.E.
5. Recommendations for tradeoffs between performance optimization and N-CET interoperability in the context of the video processing modules running on heterogeneous architectures in order to meet real time performance targets.
6. Recommendations for parameter choices based on moving object detection, blob detection and tracking performance requirements using different types of performance analysis and metrics.
7. Provide a sequential version of the registration algorithm that is capable of producing improved mosaic quality under wider operating conditions than the current algorithm being used in N-CET. Initial parallelization of the block-based matching sequential algorithm subject to contract time constraints.

There were several primary publications resulting from the work completed in the project. The first paper under review forms the basis for this report. The other two have been published or presented:

1. Praveen Kumar, K. Palaniappan, Ankush Mittal, G. Seetharaman "Parallel implementation of video processing algorithms on the multicore Cell/B.E. processor", J. Real-Time Image Processing, 2011, Under revision.
2. K. Palaniappan, I. Ersoy, G. Seetharaman, S.R. Davis, P. Kumar, R. M. Rao, R. Linderman, Parallel flux tensor analysis for efficient moving object detection, 14th Int. Conf. Information Fusion, Chicago, 2011.
3. K. Palaniappan, I. Ersoy, G. Seetharaman, S.R. Davis, R. M. Rao, R. Linderman, Multicore energy efficient flux tensor for video analysis, IEEE Workshop on Energy Efficient High-Performance Computing (HiPC), Goa, India, 2010.

2. INTRODUCTION

The continuous increase in imaging sensor data rates and increasing computational complexity of video processing algorithms for real-time motion activity analysis applications has created a pressing need for embedded high-performance computing solutions. The emergence of heterogeneous multicore architectures like the Cell Broadband Engine (Cell/B.E.) processor provides an appealing platform to explore algorithm transformations that extract data level parallelism to reach real-time performance on video streams. However, the potential benefits of such multicore processors with architecture specific data paths can be efficiently harnessed only by developing fine-grained, vectorized, embedded parallelization strategies along with algorithmic innovations. We developed parallel implementations of a typical video processing pipeline for moving object extraction consisting of several stages including flux tensor based moving object detection, binary morphological operations for noise filtering, connected component labeling (CCL) for object labeling and blob statistics for object tracking. We describe novel parallelization approaches using fine-grained optimization techniques for fully exploiting the multicore architecture and compute performance of the heterogeneous processors Synergistic Processing Elements (SPEs) on the Cell/B.E. processor. Experimental results show speedups ranging from a factor of over 600 for binary morphology to a factor of 7 for mixture of Gaussians and 5 for CCL in comparison to equivalent sequential implementations.

The recent emergence of multicore processors enables a new trend of parallel computing for achieving real-time implementations of image and video processing algorithms. New architectures and parallelization strategies are being developed for several video processing applications due to the increased accessibility of multicore, multi-threaded processors along with general purpose graphics processing units. The explosive growth in data volumes for such applications and the increasing complexity of state-of-the art algorithms necessitates a tighter optimization between utilizing new architectures and parallelization strategies for optimizing the efficient use of multicore processors. Using multi-core processors such as the IBM Cell/B.E. offers a number of advantages including low cost for the SONY PS-3 hardware solution, scalability to a larger number of cores using the IBM QS22 Blade for handling higher video processing workloads (but at higher cost per core), low power consumption and compact footprint for embedded sensor pod systems such as those envisioned for N-CET.[30] Net-centric exploitation of video sensors requires sensor management and control, monitoring and managing data collection for on-board sensors, efficient communication protocols for publishing data streams for processing as shown in Figure 1.

The N-CET project and its associated sensor pod hardware and software intelligence were designed to evaluate the tradeoffs in: 1) optimal sensor positioning, 2) reducing the need to modify aircraft avionics, and 3) broad sensor data sharing by avoiding stove-piping or one-of architectures. N-CET looks at survivability to operate close-in, ease of fielding, and reconfigurability.[30] Each N-CET node provides various information brokering services including data management, data federation among nodes, fusion of remote information with local information, and sensor control.

Automated video activity analysis has a wide range of applications, such as a target tracking, homeland security, infrastructure monitoring, urban traffic surveillance and recognizing activity patterns, etc.[8, 13, 36, 39] Video activity analysis algorithms represent a class of video processing methods that are computationally, data and bandwidth intensive. A good analysis of video surveillance workload can be found in.[22] Obtaining the desired processing frame rate of 20 to 30 frames per second (fps or fr/sec) for such algorithms in realtime, especially for high-definition video (HD), is one of the critical challenges faced by the N-CET project.

Recent multicore processors can potentially serve the needs of such workloads. For example, IBM's Cell Broadband Engine (Cell/B.E.) is based on an architecture made of eight SPEs delivering an effective peak performance of more than 200 gigaflops (GFlops), using very wide data-paths and memory interchange mechanisms. A good exposition of scientific computing and programming on the Cell/B.E. is provided in.[10] Details of implementing scientific computing kernels and programming the memory hierarchy can be found in[44] and,[16] respectively. The potential benefits of multicore processors can only be harnessed efficiently by developing parallel implementations optimized for execution on individual processing elements requiring explicit handling of data transfers and memory management. The process of refactoring legacy code and algorithms – originally optimized for sequential architectures – to modern multicore architectures invariably

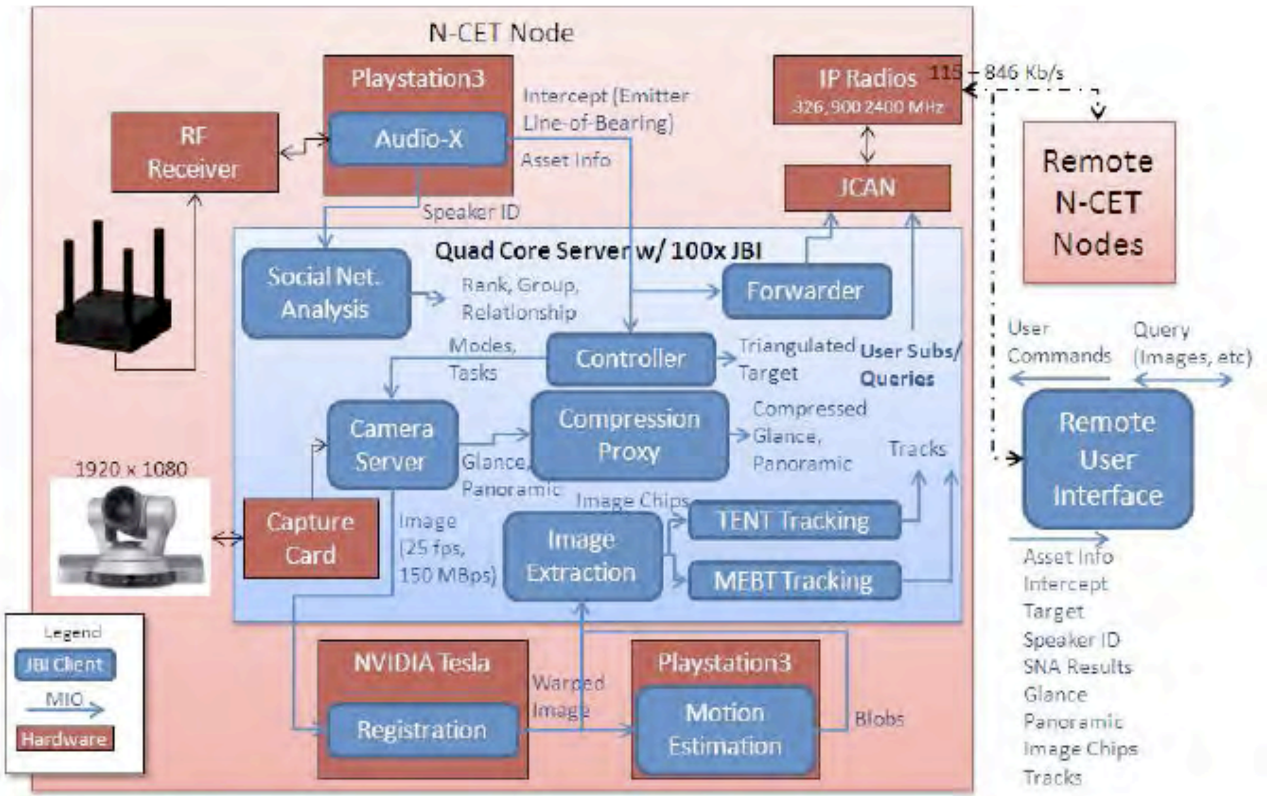


Figure 1: N-CET sensor node overall system architecture including gimbaled high-definition video camera and a directional RF antenna sensors on the left, the head processing node in the center running 100X JBI, PS-3 Cell/B.E. nodes for video processing, Joint Capability for Airborne Networking (JCAN) for inter-N-CET node communication and other off-node communications links. Figure from Metzler, et al.[30]

requires insight and reanalysis which opens the door for creative innovations in algorithm design, data structures and application specific strategies.

Research efforts from both the academia and industry have demonstrated the suitability of Cell/B.E. in video processing and retrieval,[25,46] compression[20,31] and other computer vision applications.[12] Programming frameworks/platforms like RapidMind,[28] Multicore Framework by Mercury,[7] CellSs,[6] StarPU,[4] etc. have also emerged to support efficient programming for multicore processors. In,[25] the authors implement background subtraction algorithm on Cell/B.E. by using task partitioning approach where the application is divided into modules and the PPE allocates and synchronizes the execution of different modules on different SPUs. The output is pipelined from one module or the SPE to another SPE as input to the next module. However, this approach is critically dependant on the scope of dividing the tasks into as many number of modules as there are free SPEs and how balanced is the load distribution on different SPEs. Our approach aims to extract data parallelism by implementing parallel algorithms that are scalable to the number of cores available for processing. There are quite a number of papers related to parallelizing video encoding algorithms in different standards like Joint Photographic Experts Group (JPEG) 2000,[20] H.264[25,31] on Cell/B.E. and some of them describe in detail the implementation of wavelet transform, arithmetic coding, Sum of Absolute Differences (SAD) based block matching[31] etc. However, there is not much detailed description on parallelizing video processing algorithms for the Cell/B.E.

In this report, we present our work on implementing video activity analysis algorithms applied at various stages of video processing for parallel execution on the multiple cores of Cell/B.E. For instance, we focus on algorithms like (1) Flux tensor for distinguishing stationary and moving objects, (2) Binary morphological operations and (3) Connected component labeling for blob extraction and statistics for blob characterization.

These algorithms have varying degree of computation, memory and data flow dependency characteristics which require specific parallelization and optimization strategies to handle algorithm and architecture issues. We present novel parallelization strategies for extracting data parallelism in the algorithms and offloading the computationally intensive task for parallel and scalable execution on the specialized cores. Fine grained optimization strategies like specialized vector SIMD instructions for instruction level parallelism, double buffering for overlapping computation with communication etc. is used wherever possible to fully exploit the parallel processing ability of the multicore Cell/B.E. architecture. The experimental results are conducted on different datasets with varying characteristics to test the scalability, performance gain, power efficiency, etc. of our implementation of various stages of the video processing chain for N-CET. Some relevant materials from the Phase I Technical Report are also included here for completeness especially the algorithm implementation details so that the extensive experimental results can be understood in the context of the architecture.

3. METHODS, ASSUMPTIONS AND PROCEDURES

3.1 Overview of the Cell/B.E. Processor Architecture

This short overview previously presented in the Phase I technical report is included here again to enable a self-contained report with important details of the Cell architecture emphasized in later sections. The basic architecture of the Cell/B.E. is shown in Figure 2. The 3.2 GHz SPEs deliver most of the computational capacity by executing two floating point instructions per cycle in a single instruction, multiple-data (SIMD) fashion and they can also be optimized for vector processing. Thus a peak performance of 25.6 single precision GFlop/s per SPE can be obtained. The SPE offers a high bandwidth interface to a direct memory access (DMA) engine that can transfer 32 GB/sec to and from the 256 KB local memory. The important point to note is that the SPE works only on the data that is in its local memory area called “local store”, which does not operate as a conventional central processing unit (CPU) cache. However the SPE local storage is a limited resource as only 256 Kbytes is available for program, stack, local bufferes and data structures. As such feeding the SPEs has more to do with program and data structure than with concern over any internal bottlenecks. A programmer explicitly writes DMA operation code to transfer data between the main memory and the local store. Rather than considering cache control and the impact of memory bandwidth, the focus is on structuring data movement within the Cell/B.E. processor to keep the SPEs busy, and dividing the application into vectorized functions to make efficient use of the SPEs. There are various levels of optimization possible for exploiting parallelism on the Cell/B.E. First thing to do is to divide the processing into several groups so that multiple SPEs can independently operate each group. Then execution can be optimized using Single Instruction Multiple Data (SIMD) instructions. The SPEs are short vector SIMD workhorses of the Cell/B.E. and posses a rich set of SIMD instructions which can operate simultaneously on 2 double precision values, 4 single precision values, 8 16-bit integers or 16 8-bit chars. Most of instructions are pipelined and can complete one vector operation in each clock cycle. Furthermore, communication and computation can be overlapped by double buffering mechanism. A data processing unit and a memory flow controller on SPE can operate simultaneously. The data processing unit processes the current data, while the memory flow controller transfers the next data from the main memory to the local store.

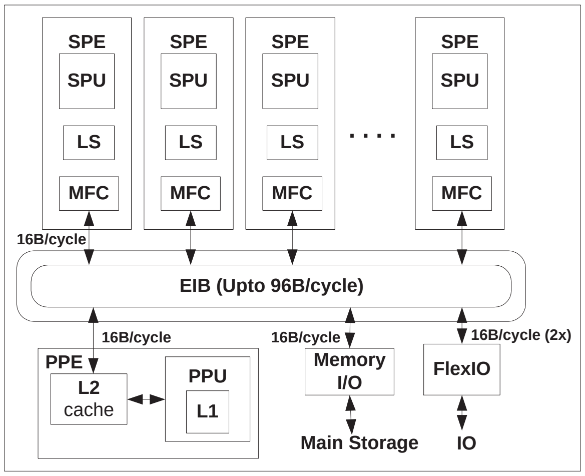
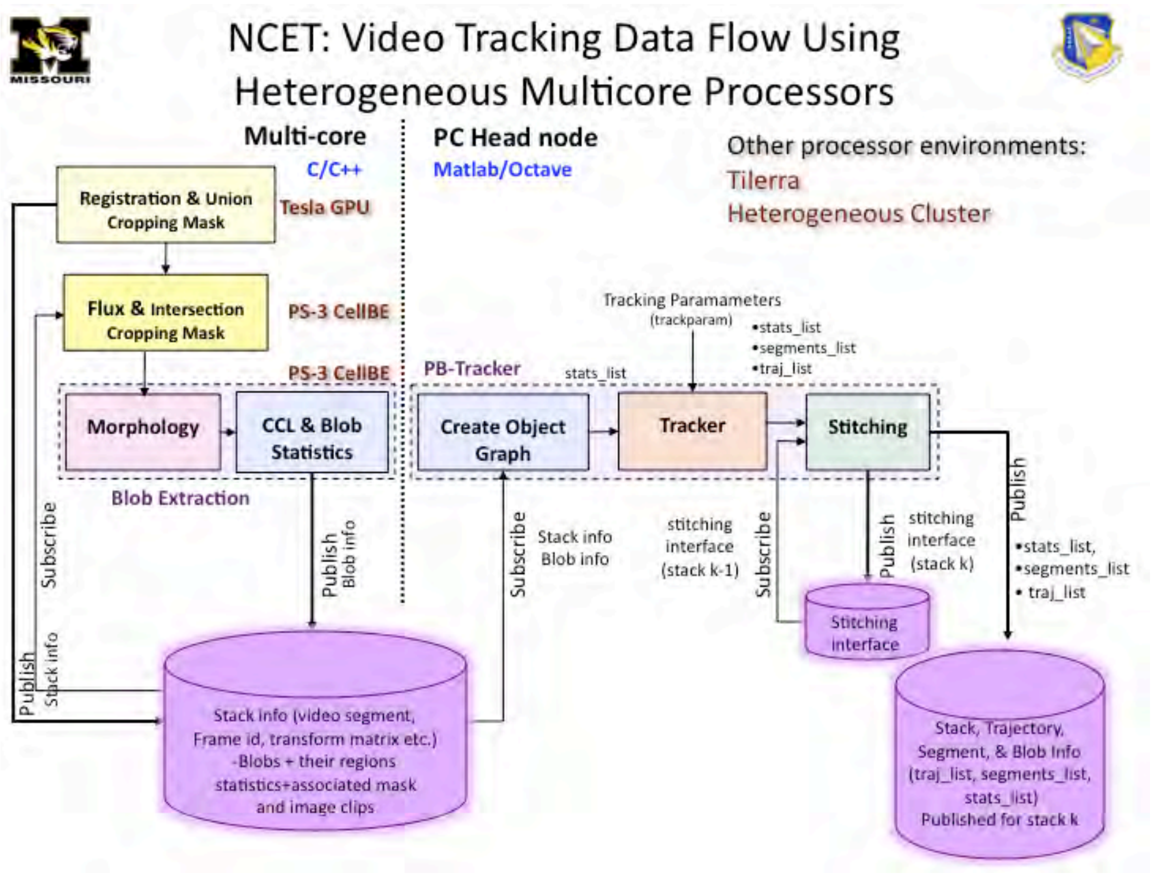


Figure 2: Overview of the Cell multicore architecture showing the nine processing cores – one Power Processing Element (PPE) with 256 MB of L2 cache and eight Synergistic Processing Elements (SPEs) without any cache but 256 KB of local store memory.

3.2 Video Motion Analysis Algorithms

The N-CET video processing chain and the interactions with the JBI/Phoenix database pub-sub services for managing large volume of image data flows is shown in Figure 3. The N-CET motion estimation client modules and groups responsible for their implementation are shown in Figure 4. Figure 5 shows the multiple



NCET: Video Tracking Data Flow Using Heterogeneous Multicore Processors
Multi-core
C/C++
PC Head node
Matlab/Octave
Other processor environments:
Tilerra
Heterogeneous Cluster
Registration & Union Cropping Mask
Tesla GPU
Flux & Intersection Cropping Mask
PS-3 CellBE
PS-3 CellBE
Morphology
CCL & Blob Statistics
Blob Extraction
PB-Tracker
Create Object Graph
stats_list
Tracking Paramameters (trackparam)
Tracker
•stats_list
•segments_list
•traj_list
Stitching
Subscribe
Publish Stack info
Publish Blob info
Subscribe
Stack info Blob info
stitching interface (stack k-1)
Subscribe
Publish
stitching interface (stack k)
Publish
•stats_list,
•segments_list
• traj_list
Stitching interface
Stack info (video segment, Frame id, transform matrix etc.) -Blobs + their regions statistics+associated mask and image clips
Stack, Trajectory, Segment, & Blob Info (traj_list, segments_list, stats_list) Published for stack k

Figure 3: N-CET video processing chain using heterogeneous multicore architectures.

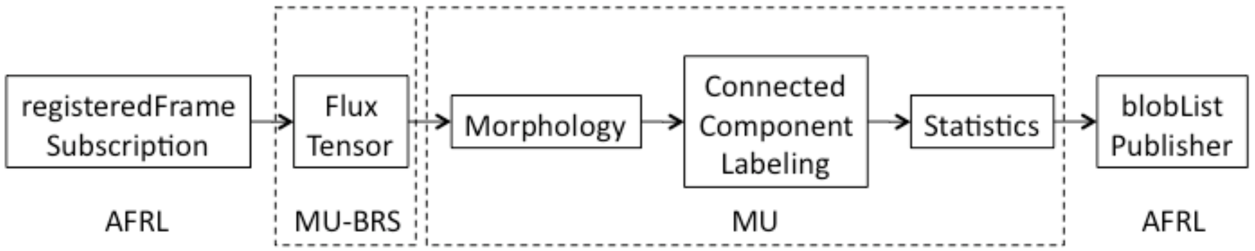


Figure 4: N-CET motion estimation client and integrated parallel pipelined workflow.

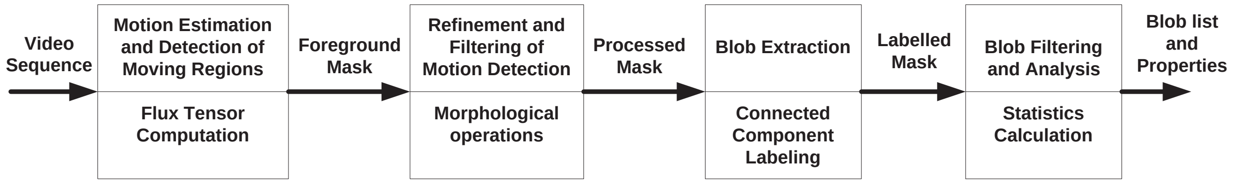


Figure 5: Overview of specific moving object detection algorithms for N-CET video motion analysis.

stages involved in typical video-based moving object detection and tracking tasks. The different stages are briefly outlined below.

1. Video frame registration. This step is required whenever the camera is not stationary for example in airborne video surveillance. The registration process occurs in three steps. First, the input video frame is divided into sub-blocks which are compared to the previous video frame to determine a motion vector for that block, and a weighting value corresponding to the expected reliability of that motion vector. Next, a least-squares fit is used to reduce this field of motion vectors to an affine transformation, and this transformation is accumulated with the previous transformation to produce the cumulative transformation over the video stream. Finally, the input video frame is transformed according to this cumulative transformation to produce the registered video frame. The most computational part of this process involves the first step of motion estimation using block matching. Although adaptive search algorithms like diamond search and efficient methods like fast normalized cross correlation, etc. have been developed, real time implementation becomes very difficult, especially for HD size images. Thus, parallel implementation of block matching algorithm is highly desirable.

2. Detection of moving foreground regions. This step forms the bulk of computation which varies depending on the complexity and robustness of the algorithm used. We use pixel-level flux tensor analysis which does not depend upon background modeling and foreground detection but uses spatiotemporal energy change to detect regions with motion. The flux tensor is an extension of the grayscale structure tensor concept and has highly structured computational organization that can be exploited in the Cell/B.E. implementation. There is high degree of data parallelism in the algorithm as it involves independent operations for every pixel making it suitable for parallelization on multicore processors. However, to meet the memory requirement of this algorithm in the limited amount of available memory within the embedded SPE cores was a challenge.

3. Consolidation, filtering and noise elimination using binary morphology. With a suitable threshold operation a binary image or mask corresponding to moving regions is created. Morphological operations "opening" and "closing" are applied to clean-up spurious responses to detach touching objects and fill in holes for single objects. Opening is erosion followed by dilation and closing is dilation followed by erosion. More details on basic erosion and dilation operator are explained in Section 3.6 in the context of parallel implementation. Opening is applied to remove small spurious flux responses, and closing would merge broken responses. There is a high degree of parallelism in this computationally expensive step as these operators have to be applied several times across the whole image. Therefore, faster parallel implementations are not only intuitively beneficial for morphological image processing but indispensable for real-time requirements.

4. Detection of connected components. In principle, the binary image resulting from Step 2 must have one connected region for each separately moving object. These regions or blobs must be uniquely labeled, in order to uniquely characterize the object pixels underlying each blob. Since there is spatial dependency at every pixel, it is not straightforward to parallelize it. Although the underlying algorithm is simple in structure, the computational load increases with image size and the number of objects − the equivalence arrays become very large and hence the processing time.[27] Furthermore, with all other steps being processed in parallel with high throughput, it becomes imperative to parallelize this step so as to avoid it from becoming a bottleneck in the processing stream. Once the connected components are detected their statistical properties such as size, bounding box, centroid, etc. are computed for input to the tracking module.

The blobs produced from the moving objects by the above algorithms are then analyzed for tracking. Blob statistics (bounding box, centroid, area, perimeter etc.) for each blob/object is calculated and used for tracking them over sequence of frames for trajectory analysis. Generally the number of objects which are candidates for tracking are not very large, this step is relatively less computationally intensive for simple tracking algorithm. Complex tracking algorithms such as multiple hypothesis tracking (MHT) are difficult to parallelize and are not dealt with in this work. The parallelization of tracking algorithms was not part of the technical scope of this project. This statistics computation can takes a significant amount of time and was parallelized in two different ways. One where the computation took place on the PPE during the relabeling stage of connected component labeling which was found to be too slow. And using a more fine-grained approach on the SPEs which sped up the computation substantially.

3.3 Registration Using Block Matching

There are several approaches for parallelizing block matching algorithms in the context of H.264 video compression[26, 31] and video retrieval applications.[46] Our intent is to present a summary of our implementation of the Sum of Absolute Differences technique in the context of video registration for completeness in the context of the N-CET motion activity analysis workload study. Although this approach was explored as part of the project we did not perform extensive experiments nor include the code as part of the final deliverable to AFRL for several reasons. An alternative GPU-based registration algorithm developed by PAR Systems under contract to AFRL was used for the N-CET video processing chain to evaluate heterogeneous architectures combining GPU and Cell/B.E. processors. The SAD implementation may not provide sufficient accuracy and quality of results desired for N-CET video processing. The SAD implementation has not been fully tested and debugged within the N-CET software environment including video streaming, sharing SPE threads, communication with JBI or Phoenix pub-sub database system for sharing results, etc.

Block matching algorithms are an important class of motion estimation algorithms, which is usually the most computationally expensive component of classical approaches to registration. These algorithms divide each frame into subblocks, called macroblocks (MBs), typically of size 16×16 pixels. The algorithm estimates the amount of motion on a block by block basis, i.e. for each block in the current frame, a block from the previous/after frame, called Reference Frame (RF) is found, that is said to match this block based on a certain criterion. Here, the block matching criteria is the Sum of Absolute Differences (SAD) and the search algorithm can be a Full Search, a three step search and a diamond search. Each block within a given search window in reference frame is compared to the current search block I(x,y). At each search step, the sum of absolute differences between the current search block and the given reference block $RF(i, j)$ is evaluated using the following criterion:

$$SAD = \sum_{i=0}^{K-1} \sum_{j=0}^{K-1} \mid RF(i,j) - MB(i - v_x, j - v_y) \mid \tag{1}$$

Where, K is the size of the block, v_x and v_y refers to the motion vector, which means amount of motion from the current macroblock to the reference block. The best match is obtained if SAD achieves the minimum. Only the intensity values (not the color information) is used in the computation, meaning that pixel values are simple unsigned character values that may be stored in one byte.

Performing a full search over all 32×32 possible locations guarantees the location with the lowest SAD value is found. But this is also computationally expensive, requiring for each point calculated 256 subtractions, 256 absolute values and 255 additions. Instead, the three step search checks eight points in a square pattern around a center pixel, re-center on the point with the lowest value, then repeat using a smaller size of square. The sizes used in pixels measured from the center are $8, 4, 2$ then 1. Diamond search is similar, but uses only four points in the cardinal directions and searches only one pixel away. It terminates when no lesser SAD value can be found.

3.4 Parallel Implementation on Cell/B.E.

In order to efficiently compute a full field of motion vectors over an entire image, the PPE partitions the frames in terms of subblocks (SBs) and distributes the computation task across multiple SPEs in parallel. Choosing a rectangular region that is a multiple of 16 in both height and width for the SB works well over a large variety of image sizes. A SPE accesses the frame data in data chunks which is composed of a row of SBs from the current frame and a row of search windows from the reference frame. This division in block of rows allows large chunk of data to be transferred into the local memory in a single DMA transaction and also allows for data reuse as the search window for the adjacent SBs overlap in the reference frame. This is illustrated in Figure 6. The left part of the figure shows the first SB (white block) in the current frame and the search window (darker gray area) in the reference frame. The right part of the figure shows the adjacent SB in the same row along with the first SB in dotted outline. This leads to a region of overlap in the search window. When all the SBs in a row are processed, the obtained results, the Motion vectors and the SAD distortion measures, are passed back to the PPE in a single DMA transaction. This process is repeated for subsequent rows of SBs allocated to the SPE. Note that there is also an overlap of data in two consecutive rows of search windows which will end up being transferred from main memory to SPE memory twice. However since the algorithm is computationally bound the extra bandwidth utilized is not a significant factor.

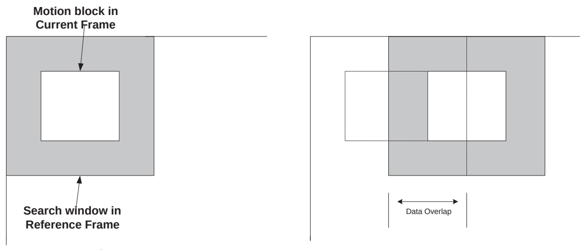

Motion block in Current Frame
Search window in Reference Frame
Data Overlap

Figure 6: Frame division in blocks of rows. The overlapping search window region for adjacent blocks in a row allows data reuse.

The Cell/B.E. processor is optimized for loading data from the local store that is aligned into 128-bit or 16 byte vector cache lines. Thus the data is packed in aligned byte vectors for optimized vector based calculations. The size of SB being a multiple of 16, allows the data from the current frame SB to be accessed in aligned 16 bytes. However, search is done by shifting pixel by pixel, requiring pixel based (misaligned) data access from the reference frame. The following code snippet shows the procedure and use of specialized instructions from SIMD intrinsic library for loading misaligned quadword on the SPE.

Listing 1: SPE code for loading misaligned quadwords

```
vec_uchar
load_qword_unaligned(vec_uchar *ptr)
{
vec_uchar qw0, qw1, qw;
unsigned int shift;
qw0 = *ptr;
qw1 = *(ptr+1);
shift = (unsigned int)(ptr) & (16-1);
qw = spu_or(spu_slqwbyte(qw0, shift,
   spu_rlmaskqwbyte(qw1, (signed)(shift -16)));
return qw;
}
```

The SAD value calculation scheme is presented in Figure 7. It is similar to the method proposed in[31] as far as the use of intrinsic instructions is concerned. The SPE SIMD intrinsic library provides two optimized vector instructions for performing: *spu_absdb* and *spu_sumb* intrinsic. The *spu_absdb* calculates the absolute value of the differences between 16 intensity values simultaneously and the *spu_sumb* sums up four by four bytes of each argument vectors in eight elements (16-bit short) of the resultant vector. 16×16 pixel macro-blocks nicely map onto these instructions because 16 128-bit vectors can represent an entire macro-block. Issuing 16 *spu_absdb* instructions followed by 8 *spu_sumb* instructions leaves 8 vectors of 8 values each. After seven more adds, the sum is accumulated to a single SAD vector of 8 values, which are finally summed into a 32-bit signed value. The sums may be computed in any order, so the algorithm is not restricted in how the data values are accessed, leaving potential for further optimization.

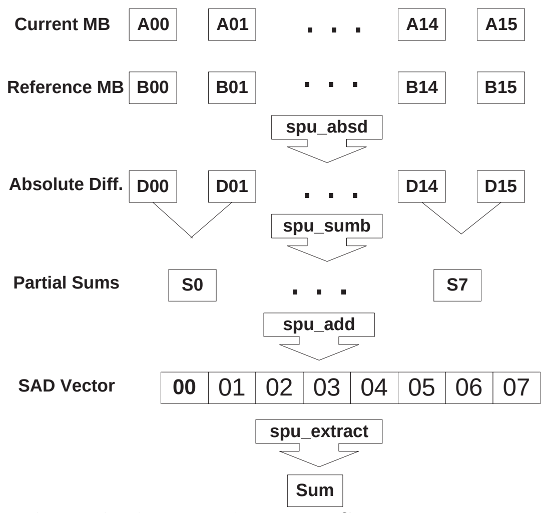


Figure 7: SAD value calculation scheme on SPU using vector instructions

3.5 Power Efficient Parallel Flux Tensor Computation for Motion Detection

The flux tensor motion flow algorithm is a versatile computer vision technique for robustly detecting moving objects in cluttered scenes. The flux tensor calculation has a high computational workload consisting of 3-D spatiotemporal filtering operations combined with 3-D weighted integration operations for estimating local averages of the flux tensor matrix trace. Power efficient real-time flux tensor processing is required in a variety of operational scenarios including video-based net-centric exploitation and tracking on airborne platforms and ground-based multi-sensor imaging for force protection. In order to achieve efficient real-time processing of high bandwidth video streams a data parallel multicore algorithm was developed for the Cell/B.E. processor and evaluated in terms of the energy to computation efficiency compared to a fast sequential CPU implementation. The Cell/B.E. has an appealing power-efficient multicore SPMD architecture for parallelizing video exploitation algorithms close-to-the-sensor. We describe the multicore parallel implementation of the dense regular computation of the flux tensor moving object detection algorithm on 3D grids, discuss architecture and algorithm specific issues and provide initial performance results. Our multicore implementation is 12 to 40 times faster than the sequential version for HD video using a single PS-3 Cell/B.E. processor and is faster than realtime for a range of filter configurations and video frame sizes. We report on the power efficiency measured in terms of performance per watt for the Cell/B.E. implementation which is 50 to 160 times better than the sequential version for HD video depending on the filter size. The results suggest an additional strategy to trade off output image quality or nominal change in accuracy of detection for improved energy efficiency in suitable environments.

Realtime persistent moving object detection and tracking for surveillance applications is a computationally challenging problem. Current trends in distributed sensor networks and agile systems favors the processing of large volumes of raw data closer to the sensor to reduce bandwidth requirements, extract high priority scene information more rapidly and exchange integrated information for cooperative downstream processing. Energy efficiency has become a leading design constraint for both hardware and software. Algorithmic approaches to improve energy efficiencies are complementary to hardware and systems-based approaches. Switching between active and sleep (or multiple low-power) modes in the idle state using a randomized algorithm for state transitions, dynamic speed-scaling using a job scheduler with fixed or flexible completion deadlines, and adaptive network topologies are several approaches that are being actively investigated for energy-efficient algorithms.[1] Multicore parallel processing environments are widely available today for which energy efficient algorithms are equally important and have greater flexibility in that the number of processors used can be dynamically changed based on a completion time versus energy tradeoff. Scheduling algorithms to minimize total energy across identical parallel processors has been shown to be non-deterministic polynomial-time hard (NP-hard) even for unit-sized jobs.[2]

We characterize the workload of the flux tensor algorithm for moving object detection in high bandwidth video streams. The parallel flux tensor algorithm exhibits super-linear speed-up due to the vectorization, loop unrolling, FMA operations and double-buffering optimizations for the Cell/B.E. architecture which along with the power efficiency of the Cell/B.E. processor provides a tremendous improvement in the performance per watt metric compared to an optimized sequential implementation. The variation in performance efficiency for different flux tensor filter sizes further suggests another avenue for energy efficient algorithm design – namely *output image quality*. That is trading off the accuracy of the flux tensor operator under certain environmental conditions using a slightly smaller filter to reduce energy use but with marginal impact on moving object detection.

Power efficient real-time flux tensor processing is required in a variety of operational scenarios including video-based net-centric exploitation and tracking on airborne platforms and ground-based multi-sensor imaging for force protection. Synergistic collaborative computation exploiting net-centricity can enable distributed interacting compute and sensor nodes to accomplish mission goals more effectively in terms of survivability, ease of fielding, and reconfigurability using a modular joint information management system[30] (see Figure 1). Such agile sensor networks need to be further enhanced to minimize overall power consumption under the constraint of still yielding the best exploitable information in a timely manner. Embedded video processing requires efficient algorithms in terms of power-aware computing as well as parallelization to enable real time performance in analyzing complex video.

There are a number of challenging computer vision problems that need to be solved for stabilizing, detecting, extracting, verifying and tracking moving objects in airborne video.[3, 14, 15, 19, 40, 47] In this section we focus on the flux tensor part of the video processing pipeline, namely power-efficient realtime moving object detection that is robust to variations in illumination, moving cast shadows or cloud movements, clutter, noise, and occlusions. In order to reliably detect moving blobs in unconstrained video, we use the recently proposed *flux tensor* (J_F) operator,[8, 9] which captures the temporal variations of the optical flow field within the local 3D spatiotemporal volume. The flux tensor detects only the moving structures, and is less sensitive to illumination, focus and related problems compared to other moving object detection algorithms including classical background subtraction, mixture of Gaussians and 3D structure tensor orientation estimation. The flux tensor motion detection results have in general better spatial coherency enabling more accurate motion-based object segmentation. The flux tensor is more efficient in comparison to the 3D grayscale structure tensor since motion information is more directly incorporated in the flux calculation which is less expensive than eigenvalue decompositions at each pixel in the image.

This section describes in detail the parallel implementation of the flux tensor optimized for the multicore Cell/B.E. processor for real-time processing of high-bandwidth video streams in power constrained environments. Some early supercomputing architectures like the SIMD MasPar were ideally suited for image analysis tasks like deformable motion estimation[34] and similar architectures are being mapped onto the embedded processor level. The PS-3 Cell/B.E. processor provides a modern power efficient single chip high performance computational platform, with seven heterogeneous cores - one Power Processing Element (PPE) and six (of eight) active Synergistic Processing Elements (SPEs). The PPE is a 64-bit processor that is binary-compliant with the PowerPC 970 but with a simpler architecture supporting dual issue, in-order execution. Each SPE consists of a 3.2 GHz Synergistic Processing Unit (SPU), a large 128-entry 128-bit vector register file, a small 256 Kbytes of private local store memory, short pipelines, and a memory-flow controller (MFC) to access the 256 MB of shared main memory using non-blocking DMA commands at 25.6 Gbytes/s. The SPUs are in-order dual-issue statically scheduled short-vector number crunchers with support for SIMD instructions operating on packed multiple data value without dynamic branch prediction. The PS-3 version of the Cell processor is optimized for single-precision arithmetic (double-precision peak is less than 11 GFLOP/s) with truncation rounding. Each SPE can perform 25.6 GFLOP/s single-precision floating point operations at 3.2GHz. Although the six SPEs can deliver 153.6 GFLOP/s peak performance, memory-intensive single-precision calculations max out at 12.8 GFLOP/s (and double-precision at 6.4 GFLOP/s) due to memory bandwidth limits.[24] The Cell/B.E. uses the single program multiple data (SPMD) parallel processing model which is more powerful than the single instruction multiple data (SIMD) model for heterogeneous multithreaded data flow execution mapped onto SPEs.

The Cell/B.E. offered one of the first commercial implementations of a power efficient high performance single chip multiprocessor with a significant number of general-purpose programmable cores targeting a broad set of workloads.[18] A good description of scientific computing and programming on the Cell is provided in[10] and other details of implementing scientific computing kernels and programming memory hierarchies can be found in.[16, 44] In,[45] the authors discuss interesting code transformation techniques for moving scientific simulation codes to the Cell/B.E. and[5] describes the fastest Fourier transform for the Cell processor (18.6 GFLOP/s). Several programming models like RapidMind,[28] Multicore Framework by Mercury,[7] CellSs,[6] StarPU[4] have also emerged to support efficient programming for multicore processors. In order to reduce complexities of task management, multithreading and synchronization for programming the Cell/B.E. some tools for mapping serial code in a semi-automatic fashion are in development.[37, 38] We first give a brief overview of the flux tensor method and discuss the sequential implementation along with the computation and memory characteristics. Then we discuss the parallel architecture issues involved in our Cell/B.E. implementation. A description of the data partitioning scheme and parallelization procedures to map the flux tensor algorithm onto the Cell/B.E. cores is followed by experimental results.

3.5.1 Flux Tensor-based Motion Detection

The 3D flux tensor was shown to be a robust and computationally efficient method for coherent detection of moving regions in video.[8, 9, 33] The flux tensor is a more efficient operator in comparison to the 3D grayscale structure tensor[35, 43] since motion information is more directly incorporated in the flux calculation without the necessity for computing eigenvalue decompositions as with the 3D grayscale structure tensor. We summarize the mathematical description of the structure tensor and flux tensor multidimensional orientation estimation methods in order to provide a background on the types of operators needed to compute the flux tensor quantity for robust motion estimation.

Under the constant illumination model, the optic-flow (OF) equation of a spatiotemporal image volume $I(\mathbf{x})$ centered at location $\mathbf{x} = [x, y, t]$ is given by Eq. 2 where, $\mathbf{v}(\mathbf{x}) = [v_x, v_y, v_t]$ is the optic-flow vector[21] at $\mathbf{x}$,

$$\begin{aligned} \frac{dI(\mathbf{x})}{dt} &= \frac{\partial I(\mathbf{x})}{\partial x}\, v_x + \frac{\partial I(\mathbf{x})}{\partial y}\, v_y + \frac{\partial I(\mathbf{x})}{\partial t}\, v_t \\ &= \nabla I^T(\mathbf{x})\, \mathbf{v}(\mathbf{x}) = 0 \end{aligned} \tag{2}$$

and $\nabla \mathbf{I}(\mathbf{x})$ is the image gradient vector field. A constrained total least-squares minimization approach can be used to estimate the spatio-temporal orientation vectors $\mathbf{v}(\mathbf{x})$. Using the normalization constraint $||\mathbf{v}(\mathbf{x})|| = 1$ in order to avoid degenerate solutions, leads to the minimization functional,

$$\begin{aligned} e_{ls}(\mathbf{x}) = \int W(\mathbf{x}, \mathbf{y}) \left(\nabla I^T(\mathbf{y})\, \mathbf{v}(\mathbf{x})\right)^2 d\mathbf{y} \\ + \lambda \left(1 - \mathbf{v}(\mathbf{x})^T \mathbf{v}(\mathbf{x})\right) \end{aligned} \tag{3}$$

with Lagrange multiplier λ, $W(\mathbf{x}, \mathbf{y}) = W(\mathbf{x} - \mathbf{y})$ is a *spatially invariant* weighting function (e.g., Gaussian) that is localized, centered at $\mathbf{x}$ and emphasizes image gradients near the central pixel[32] and the integral is a multidimensional integral. Assuming a constant orientation $\mathbf{v}(\mathbf{x})$ model within the neighborhood $W(\mathbf{x}, \mathbf{y})$ solving for the minimum of $e_{ls}(\mathbf{x})$ in (Eq. 3) leads to the eigenvalue problem,

$$\mathbf{J}(\mathbf{x}, W)\, \hat{\mathbf{v}}(\mathbf{x}) = \lambda\, \hat{\mathbf{v}}(\mathbf{x}) \tag{4}$$

where $\hat{\mathbf{v}}(\mathbf{x})$ is the best spatiotemporal orientation estimate of $\mathbf{v}(\mathbf{x})$, and $\mathbf{J}(\mathbf{x}, W)$ the 3D structure tensor matrix associated with the spatiotemporal volume centered at $\mathbf{x}$ is shown below using an outer product notation,

$$\mathbf{J}(\mathbf{x}, W) = \int W(\mathbf{x} - \mathbf{y}) \left(\nabla I(\mathbf{y})\, \nabla I^T(\mathbf{y})\right) d\mathbf{y} \tag{5}$$

The structure tensor matrix $\mathbf{J}(\mathbf{x}, W)$ is symmetric, real and composed of first-order derivatives with each matrix element given by,

$$J_{pq}(\mathbf{x}, W) = \int W(\mathbf{x}-\mathbf{y}) \left(\frac{\partial I(\mathbf{y})}{\partial x_p} \frac{\partial I(\mathbf{y})}{\partial x_q} \right) d\mathbf{y} \tag{6}$$

The elements of $\mathbf{J}$ (Eq. 6) describes information relating to local spatial and temporal gradients in simple neighborhoods. A typical approach in motion detection is to threshold $\mathbf{Trace}(\mathbf{J})$,

$$\begin{aligned}\mathbf{Trace}(\mathbf{J}(\mathbf{x})) &= \int W(\mathbf{x}-\mathbf{y}) ||\nabla I||^2 d\mathbf{y} \\ &= \int W(\mathbf{x}-\mathbf{y}) (\frac{\partial I}{\partial x})^2 d\mathbf{y} + \int W(\mathbf{x}-\mathbf{y}) (\frac{\partial I}{\partial y})^2 d\mathbf{y} \\ &+ \int W(\mathbf{x}-\mathbf{y}) (\frac{\partial I}{\partial t})^2 d\mathbf{y}\end{aligned} \tag{7}$$

but this results in ambiguities in distinguishing responses arising from stationary versus moving features (e.g., edges and junctions with and without motion), since $\mathbf{Trace}(\mathbf{J})$ incorporates total gradient change information but fails to capture the nature of these gradient changes (i.e. spatial only versus spatiotemporal).[42] To resolve this ambiguity and to classify the video regions experiencing motion, the eigenvalues and the associated eigenvectors of $\mathbf{J}$ are usually analyzed.[9, 35, 42, 48] Motion associated with small eigenvectors are classified as noise and discarded. However, eigenvalue decomposition at every pixel is computationally expensive especially if real time performance is required. In order to reliably detect only the moving structures *without* performing expensive eigenvalue decompositions, the *flux tensor* has been shown to be a more robust operator.[8, 33] The flux tensor is composed of the temporal variations in the optical flow field within the local 3D spatiotemporal volume. Computing the second derivative of Eq. 2 with respect to t, Eq. 16 is obtained where, $\mathbf{a}(\mathbf{x}) = [a_x, a_y, a_t]$ is the acceleration of the image brightness located at $\mathbf{x}$.

$$\begin{aligned}\frac{\partial}{\partial t}\left(\frac{dI(\mathbf{x})}{dt}\right) &= \frac{\partial^2 I(\mathbf{x})}{\partial x \partial t} v_x + \frac{\partial^2 I(\mathbf{x})}{\partial y \partial t} v_y + \frac{\partial^2 I(\mathbf{x})}{\partial t^2} v_t \\ &+ \frac{\partial I(\mathbf{x})}{\partial x} a_x + \frac{\partial I(\mathbf{x})}{\partial y} a_y + \frac{\partial I(\mathbf{x})}{\partial t} a_t\end{aligned} \tag{8}$$

which can be written in vector notation as,

$$\frac{\partial}{\partial t}(\nabla I^T(x)\mathbf{v}(\mathbf{x})) = \frac{\partial \nabla I^T(\mathbf{x})}{\partial t} \mathbf{v}(\mathbf{x}) + \nabla I^T(\mathbf{x})\, \mathbf{a}(\mathbf{x}) \tag{9}$$

Using the same approach for deriving the classic 3D structure, minimizing Eq. 16 assuming a constant velocity model and subject to the normalization constraint $||\mathbf{v}(\mathbf{x})|| = 1$ leads to,

$$\begin{aligned}e^F_{ls}(\mathbf{x}) &= \int W(\mathbf{x}, \mathbf{y}) \left(\frac{\partial \nabla I^T(\mathbf{y})}{\partial t} \mathbf{v}(\mathbf{x}) \right)^2 d\mathbf{y} \\ &+ \lambda \left(1 - \mathbf{v}(\mathbf{x})^T \mathbf{v}(\mathbf{x}) \right)\end{aligned} \tag{10}$$

Using a constant velocity model within a local neighborhood $\mathbf{\Omega}(\mathbf{x}, \mathbf{y})$, results in the acceleration field being zero everywhere. As with its 3D structure tensor counterpart $\mathbf{J}$ in Eq. 5, the 3D flux tensor $\mathbf{J_F}$ based on the total least squares solution of Eq. 10 can be written as,

$$\mathbf{J_F}(\mathbf{x}, W) = \int W(\mathbf{x}, \mathbf{y}) \frac{\partial}{\partial t} \nabla I(\mathbf{x}) \cdot \frac{\partial}{\partial t} \nabla I^T(\mathbf{x}) d\mathbf{y} \tag{11}$$

which can be expanded to the matrix form,

$$\mathbf{J_F} = \begin{bmatrix} \int W \{\frac{\partial^2 I}{\partial x \partial t}\}^2 d\mathbf{y} & \int W \frac{\partial^2 I}{\partial x \partial t} \frac{\partial^2 \mathbf{I}}{\partial y \partial t} d\mathbf{y} & \int W \frac{\partial^2 I}{\partial x \partial t} \frac{\partial^2 \mathbf{I}}{\partial t^2} d\mathbf{y} \\ \int W \frac{\partial^2 I}{\partial y \partial t} \frac{\partial^2 \mathbf{I}}{\partial x \partial t} d\mathbf{y} & \int W \{\frac{\partial^2 I}{\partial y \partial t}\}^2 d\mathbf{y} & \int W \frac{\partial^2 I}{\partial y \partial t} \frac{\partial^2 \mathbf{I}}{\partial t^2} d\mathbf{y} \\ \int W \frac{\partial^2 I}{\partial t^2} \frac{\partial^2 \mathbf{I}}{\partial x \partial t} d\mathbf{y} & \int W \frac{\partial^2 I}{\partial t^2} \frac{\partial^2 \mathbf{I}}{\partial y \partial t} d\mathbf{y} & \int W \{\frac{\partial^2 I}{\partial t^2}\}^2 d\mathbf{y} \end{bmatrix} \tag{12}$$

As seen from Eq. 12, the elements of the flux tensor incorporate information about temporal gradient changes which leads to efficient discrimination between stationary and moving image features. Thus the trace of the flux tensor matrix can be compactly written and computed as,

$$\mathbf{Trace}(\mathbf{J_F}(\mathbf{x})) = \int W(\mathbf{x}-\mathbf{y})||\frac{\partial}{\partial t}\nabla I||^2 d\mathbf{y} \tag{13}$$

and can be directly used to classify moving and non-moving regions without the need for expensive eigenvalue decompositions. The trace of the flux tensor matrix, referred to as Tr_J_F, can be written explicitly as,

$$\mathbf{Trace}(\mathbf{J_F}) \quad = \quad \int W(\mathbf{x}-\mathbf{y})\{\frac{\partial^2 I}{\partial x \partial t}\}^2 d\mathbf{y} \; + \; \int W(\mathbf{x}-\mathbf{y})\{\frac{\partial^2 I}{\partial y \partial t}\}^2 d\mathbf{y} \; + \; \int W(\mathbf{x}-\mathbf{y})\{\frac{\partial^2 I}{\partial t \partial t}\}^2 d\mathbf{y} \tag{14}$$

where W is a spatially invariant weighted averaging operator. Using the simplified derivative notation,

$$I_{xt} = \frac{\partial^2 I}{\partial x \partial t}, \;\; I_{yt} = \frac{\partial^2 I}{\partial y \partial t}, \;\; I_{tt} = \frac{\partial^2 I}{\partial t \partial t} \tag{15}$$

then Eq. 17 requires calculating the second derivatives I_{xt}, I_{yt} and I_{tt} followed by numerically integrating the squares of I_{xt}, I_{yt} and I_{tt} within the spatiotemporal window W. Robust statistics methods can be used to scale and compare the flux tensor trace response in a data adaptive manner while minimizing the use of fixed thresholds for determining which pixels belong to the set of moving objects.[49, 50]

To summarize, in order to reliably detect moving structures *without* performing expensive eigenvalue decompositions, the *flux tensor* has been shown to be a more robust operator in comparison to the more widely used structure or orientation tensor.[8, 33] The flux tensor is composed of the temporal variations in the optical flow field within the local 3D spatiotemporal volume. Computing temporal derivative of the optical flow equation and setting the image brightness acceleration to zero gives,

$$\frac{\partial}{\partial t}\left(\frac{dI(\mathbf{x})}{dt}\right) = I_{xt}\, v_x + I_{yt}\, v_y + I_{tt}\, v_t, \tag{16}$$

where $I(\mathbf{x})$ is the spatiotemporal image volume, t is time, $\mathbf{v}(\mathbf{x}) = [v_x, v_y, v_t]$ is the optic-flow vector at $\mathbf{x}$, and the second derivative terms are defined as before. The I_{xt} and I_{yt} terms capture information about moving edges or gradients in the video while I_{tt} incorporates information on moving textures and temporal illumination changes. A total least squares solution to Eq. 16 leads to the structure tensor matrix, $\mathbf{J_F}(\mathbf{x}, W(\mathbf{x}, \mathbf{y}))$, with an integration kernel $W(\mathbf{x}, \mathbf{y})$. We use the trace of the flux tensor matrix, referred to as Tr_J_F, that is defined below,

$$Tr\_J_F = \int_\Omega W(\mathbf{x}-\mathbf{y})(I_{xt}^2(\mathbf{y}) + I_{yt}^2(\mathbf{y}) + I_{tt}^2(\mathbf{y}))d\mathbf{y} \tag{17}$$

as the computational operator to reliably detect moving regions in video streams. A spatially invariant integration kernel $W(\mathbf{x}-\mathbf{y})$, also referred to as the local averaging operator, is used for low power operation (instead of a more expensive spatially varying kernel) and is applied after the derivative computations in the flux tensor trace are completed.

3.5.2 Numerical Computation of the Flux Tensor

The calculation of the second derivative operators needed to compute the trace of the flux tensor matrix are implemented as convolutions with appropriate kernel filters. Although general 3D convolution kernels can be used, separable kernels are preferred as the 3D convolutions then can be decomposed into a cascade of 1D convolutions with a substantial reduction in computational cost from $O(n_k^3)$ to $O(n_k)$ for an $n_k \times n_k \times n_k$ sized filter. For numerical stability as well as noise reduction, a smoothing filter is applied along the third dimension that is not involved in the specific second derivative filter. The calculation of the first component of the trace, I_{xt}, uses derivative filters in the x- and t-dimensions and smoothing along the y-dimension, whereas calculation of I_{yt} uses smoothing along the x-dimension. The final component of the flux tensor matrix trace, I_{tt}, is the second derivative along the temporal direction and in this case the smoothing

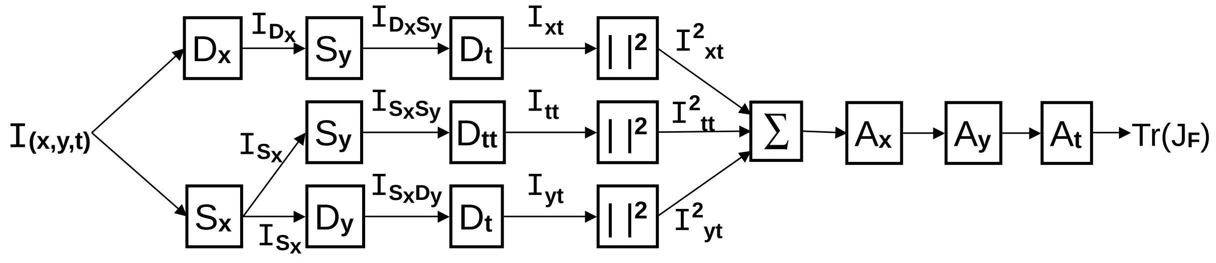


Figure 8: Operator-centric data flow view of the various stages required to compute the flux tensor operator on a 3D spatiotemporal volume showing the task dependency relationships. Note that the magnitude squaring operators are explicitly shown. The summation stage is done prior to spatiotemporal averaging steps for computational efficiency at the cost of reduced parallelism.

is applied along both spatial dimensions. The integral operator is also implemented numerically as an averaging filter decomposed into three 1D filters. The operation flow is illustrated in Figure 8. The data flow objects $I_{D_xS_y}$, $I_{S_xS_y}$ and $I_{S_xD_y}$ represent the intermediate spatial convolution results required to calculate I_{xt}, I_{yt} and I_{tt}. The operator modules shown in Figure 8, are the spatial smoothing filters S_x and S_y, the spatial derivative filters D_x and D_y, both in the x- and y-directions respectively, and the temporal derivative operators D_t and D_{tt}, representing first and second derivative filters in t respectively. The final averaging filters are the integral part of the flux tensor operator with A_x, A_y and A_t representing averaging filters in x-, y- and t-directions respectively. The data flow shown in Figure 8 reflects optimizations for a sequential implementation. Specifically, the summation block is being done prior to the spatiotemporal averaging operators for improved computational efficiency but at the expense of increased task dependencies and reduced parallelism (see Eq. 18 discussion). Exchanging the order of the sum and averaging filters will increase parallelism but would require more memory or additional computation. For the implementation shown in Figure 8, calculating the flux tensor trace for each video pixel requires eight 1D convolutions for the three spatiotemporal derivatives and three 1D convolutions for local averaging filters within the corresponding spatiotemporal cubes. The number of temporal filtering operations is reduced by saving intermediate results using additional memory.

The filter lengths or tap sizes associated with the three kernels for computing the flux tensor trace, $(n_{S_x}, n_{S_y}, n_{D_x}, n_{D_y},\ n_{D_t}, n_{D_{tt}}, n_{A_x}, n_{A_y}, n_{A_t})$ are the full set of filter parameters that would need to be specified for a given application. Since we use spatially isotropic filters, we have a reduced set of parameters to specify, with $n_{D_x} = n_{D_y} = n_{D_s}, n_{S_x} = n_{S_y} = n_{S_s}, n_{A_x} = n_{A_y} = n_{A_s}$. Typically we use the same filter lengths for the first and second temporal derivative kernels ($= n_{D_t}$) and for the spatial smoothing and derivative kernels, *i.e.*, $n_{S_s} = n_{D_s}$. Thus, there remains four main parameters of the flux tensor; $(n_{D_s}, n_{D_t}, n_{A_s}, n_{A_t})$ which are the 1D filter sizes of the spatial derivative filter, the temporal derivative filter, the spatial averaging filter, and the temporal averaging filter respectively. In medium to close view shots, the choice of $(5, 5, 5, 5)$ for filter sizes works well for detection. For the very far view sequences, where the objects may be quite small and moving very slowly, a $(3, 9, 3, 3)$ size works best. The large temporal filter size helps to catch the slow motion, the small spatial filter size helps to detect small motion and keeps the smoothing to a minimum.

3.5.3 Sequential Implementation of the Flux Tensor Operator

The flux tensor implementation uses just the luminance component of the RGB video (1920×1080 pixels). In our earlier work,[9] we described a reference sequential implementation that has minimum memory requirement and used just a single input image First In First Out (FIFO) buffer of size $(n_{D_t} + n_{A_t} - 1)$ for storing the input frames but at the cost of recomputing all spatiotemporal derivatives and integrals for each new video frame. This can be a significant penalty in terms of time and power since many intermediate filtering results that can be reused now have to be recomputed. A more efficient sequential implementation that minimizes redundant computations using one larger FIFO buffer of size $4 * (n_{D_t} + n_{A_t} - 1)$, for the intermediate spatial and temporal derivatives, and storing these intermediate results to be reused across temporal stages is described previously.[8] Here, we discuss a new alternative approach that further improves memory efficiency by using dual FIFO buffers with a smaller memory footprint of $3 * n_{D_t} + n_{A_t}$ plus a few additional frames. Each frame of the input sequence is first convolved with spatial derivatives and smoothing filters. The intermediate results are stored as frames to be used in temporal convolutions, and pointers to these frames are stored in a FIFO buffer. The size of the first FIFO structure is of length n_{D_t} and for each input frame three spatial derivative frames $I_{D_x S_y}, I_{S_x D_y}$ and $I_{S_x S_y}$ are calculated and stored. Hence, the number of frames that need to be stored in the first FIFO structure is $3n_{D_t}$. Once n_{D_t} frames are processed and stored, the FIFO structure has enough frames for calculation of the temporal derivatives. Three frames of storage are needed to hold the temporal derivatives in memory for the current timestep.

Since averaging is distributive over addition for linear operators, the sum of squares $I_{xt}^2 + I_{yt}^2 + I_{tt}^2$, which is the trace of the flux tensor matrix is computed first, then spatial averaging is applied to this result and stored in a second FIFO structure of size n_{A_t}, to be used in the temporal part of averaging. The numerical expression that is being computed is,

$$Tr\_J_F(\mathbf{x}) = \sum_{\mathbf{y} \in \mathcal{N}(x,y,t)} W(\mathbf{x} - \mathbf{y}) \big(I_{xt}^2(\mathbf{y}) + I_{yt}^2(\mathbf{y}) + I_{tt}^2(\mathbf{y}) \big) \tag{18}$$

where $\mathcal{N}$ is the local neighborhood over which the square of the second derivatives are summed. A weighted averaging filter, such as a Gaussian, can be used at the expense of additional computing cost. Typically box filters are used for a power efficient implementation. This temporal averaging FIFO keeps n_{A_t} frames at a time and produces the flux tensor trace after it is full. Once both FIFO's are full, processing a new input frame causes a shift of pointers in both FIFO's, reusing intermediate results from previous calculations and reducing the total computation per flux tensor output frame. Figure 9 shows the steady state operation of FIFO's for $n_{D_t} = 5$ and $n_{A_t} = 5$.

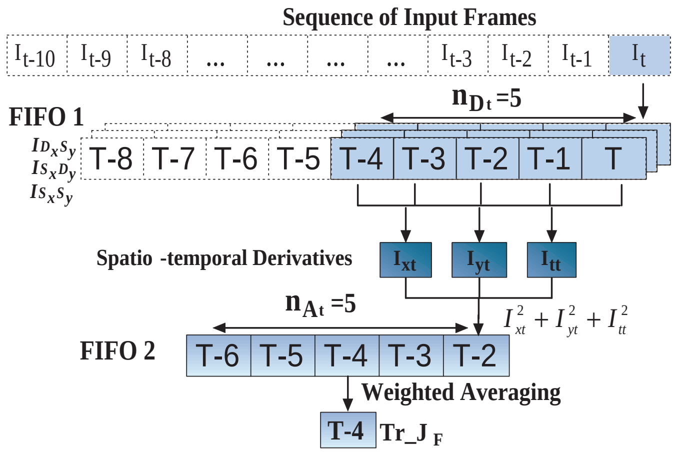


Figure 9: Steady state flow in sequential flux tensor implementation. The memory footprint (MF) for the given filter sizes ($n_{D_t} = 5, n_{A_t} = 5$) is $MF = 25 \times M \times N \times 4$ bytes.

3.5.4 Parallel Implementation of the Flux Tensor Operator on the Cell/B.E.

A power efficient multicore implementation needs to take full advantage of the intrinsic Cell/B.E. architecture specific hardware accelerations in order to use the best choice of data and task partitioning across SPEs, managing memory transfers, taking full advantage of vectorization and using local buffers. The 3.2 GHz

SPEs deliver their peak performance while executing a fused short vector multiply add instruction (FMA) on each clock cycle that operates on a four floating-point element vector to complete eight floating point operations in SIMD fashion. Thus a peak performance of 25.6 single-precision GFLOP/s per SPE can be obtained. An important point to note is that the SPEs work only on data that is in its local memory (local store). However, the SPE local storage is a limited resource as only 256 Kbytes is available for program, stack, local buffers and data structures. Making sure the SPEs efficiently receive and operate on current data without excessive buffering is critical to achieving high performance on the Cell/B.E. architecture. Rather than considering cache control and the impact of memory bandwidth, we focus on structuring data movement within the Cell/B.E. processor to keep the SPEs busy, and dividing the application into vectorized functions to make efficient use of the SPE hardware.

We implemented and tested our code on the SONY PS-3 which is a very energy efficient multicore processor but only six of eight Cell SPE processors are available and the main/external XDR memory on the PPE is a restrictive 256 MB of which only about 200 MB is available to the Linux OS. To accommodate the required frame storage buffers and data structures for the parallel implementation, the data partitioning and grouping of the operations needed careful consideration. For main memory the parallel algorithm uses two FIFO buffers of size n_{D_t} and n_{A_t} for calculation of temporal derivatives and temporal averaging. In the sequential implementation, all the spatial derivatives are kept in the first FIFO buffer which requires $3n_{D_t}$ frames to be stored in main memory. However, there is insufficient shared (global) memory on the Cell/B.E. to cache this many intermediate results. Additionally, there is significant communication overhead increasing the communication-to-compute ratio, since the intermediate results need to be transferred back and forth between main memory and the SPE local store, which reduces overall efficiency. Consequently, for the Cell/B.E. parallel implementation of the flux tensor, we store only the input sequence of images in the first FIFO buffer of size n_{D_t} (instead, of $3n_{D_t}$ for storing the spatial derivatives) and recalculate the spatiotemporal derivatives for each successive frame at the cost of redundant calculations. We estimate the amount of redundant work to be between a factor of 2 and 5 depending on the filter sizes shown in Table 1, compared to the sequential version. There is no overlap in the work unit computation between adjacent SPEs, as shown by the vertical lines and faces marked in red in Figure 11 but there is an extra quad word data transfer (16 bytes) on left and right sides to provide pixel padding in the x-convolution direction. The second buffer FIFO2 operates the same as in the sequential case. The steady state behavior of the FIFO's for the Cell/B.E. implementation is shown in Figure 10.

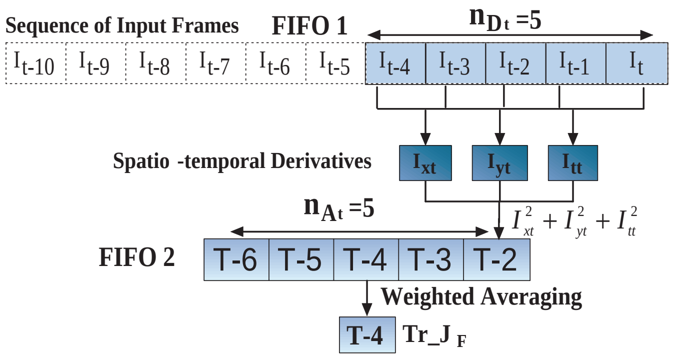

Sequence of Input Frames
FIFO 1
$n_{D_t}=5$
I_{t-10} I_{t-9} I_{t-8} I_{t-7} I_{t-6} I_{t-5} I_{t-4} I_{t-3} I_{t-2} I_{t-1} I_t
Spatio -temporal Derivatives
I_{xt} I_{yt} I_{tt}
$I_{xt}^2 + I_{yt}^2 + I_{tt}^2$
$n_{A_t}=5$
FIFO 2
T-6 T-5 T-4 T-3 T-2
Weighted Averaging
T-4
Tr_J_F

Figure 10: Steady state flow in flux tensor implementation on Cell/B.E.. The memory footprint for the given filter sizes ($n_{D_t} = 5, n_{A_t} = 5$), $MF = 14 \times M \times N \times 4$ bytes is much smaller than for the sequential version where there is often a large amount of fast memory available for the processor cores.

In order to parallelize across SPEs, the data needs to be partitioned into equal work blocks amongst different SPEs for optimal performance. Since there are convolutions in three dimensions (x, y, t), the whole work unit can be visualized as a 3D block. This is partitioned into as many overlapping blocks as there are number of active SPEs. The data is fetched and processed one row at a time. Due to finite size of the local store memory, each SPE may further subdivide the work block into smaller chunks and process

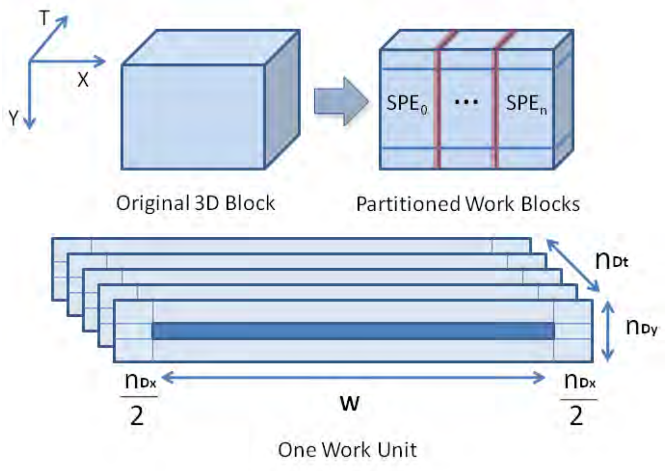


Figure 11: Data partitioning scheme showing Work Unit (WU) block width in pixels on SPEs. The 3D block represents the spatiotemporal 3D grid of input data that needs to be processed to produce one flux tensor output frame. The 3D grid of data is chunked uniformly to distribute to each SPE. Since the partition is too large to fit into the limited SPE memory, each work block is further divided into smaller work units. The WU sizes are dependent on the filter sizes as labeled.

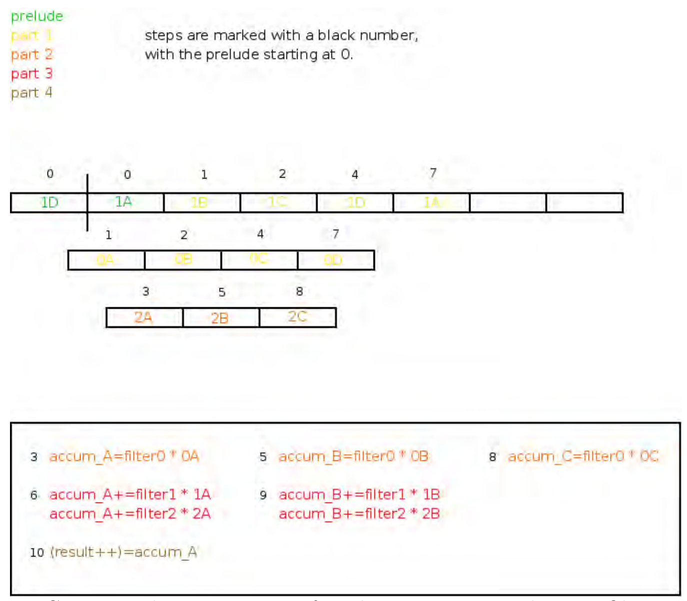


Figure 12: Efficient SPE implementation of a three tap convolution filter in the x-direction.

a work unit width of WU columns each time. The data partitioning scheme and full work unit block is illustrated in Figure 11. The execution process on the PPE and SPE side is summarized in Algorithms 1 and 2 respectively. Optimized convolution operators are represented using the $\otimes$ symbol, without the explicit loop unrolling or optimized FMA operations are explicitly shown for simplicity. The efficient implementation of the convolution operators is shown in Figure 12.

Algorithm 1 Parallel Flux Tensor: PPE side

```
Input : Input Image sequence $I(x, y, t)$
Output : Flux Trace frame $Tr\_J_F(x, y, t - \lfloor n_{D_t}/2 \rfloor - \lfloor n_{A_t}/2 \rfloor)$
1: for each time $t$ do
2:   Push($I(x, y, t)$, FIFO1)
3:   Initialize number of intermediate flux frames, $N_m \leftarrow 0$
4:   if FIFO1 contains $n_{D_t}$ frames then
5:     Partition data into blocks.
6:     Put SPE control block information including work unit $W$ and output location for intermediate flux $F$ and
       final output $Tr\_J_F$.
7:     Set up SPE threads and wait for results $F$, $Tr\_J_F$.
8:     Push($F$, FIFO2)
9:     $N_m \leftarrow N_m + 1$
10:    if $N_m > n_{A_t}$ then
11:      Write output $Tr\_J_F$
12:    end if
13:  end if
14: end for
```

3.5.5 Theoretical Estimation of the Flux Tensor Computational Requirements

The steady state operation of FIFO's for $n_{D_t} = 5$ and $n_{A_t} = 5$ is shown in Figure 9. The memory footprint (MF) in number of frames and the delay or lag (ℓ) between input and output are given as,

$$MF = 3n_{D_t} + n_{A_t} + 5, \;\; \ell = \lfloor (n_{D_t} + n_{A_t})/2 \rfloor - 1 \tag{19}$$

Our initial theoretical estimates for the number of multiplications (N_m) required to calculate the flux response of a frame of $N \times M$ pixels for a given filter sizes $(n_{D_s}, n_{D_t}, n_{A_s}, n_{A_t})$ was,

$$N_m = MN(3n_{D_t} + n_{A_t} + 2n_{A_s} + 6n_{D_s} + 3) + (M + N)[n_{A_s}(1 - n_{A_s}) + 3n_{D_s}(1 - n_{D_s})] \tag{20}$$

For $(n_{D_s}, n_{D_t}, n_{A_s}, n_{A_t}) = (7, 5, 7, 5)$ and $M = 640, N = 480$ image size then $N_m = 24,268,800$ multiplications (and an equivalent number of additions) per frame. For 30 fps performance the flux tensor operator would require 1.5 GFLOPs (728 megaflops (MFlops) each for multiply and add) and for HD about 330 MFlops per frame for the sequential algorithm.

Further analysis, led a more refined estimate for the theoretical estimate of the computational cost for the flux tensor implementation. Using a consistent notation between the report and the earlier work we gave the following calculations. Following are the old notation and their new counterparts:

$N_{sx} = N_{sy} = N_{dy} = N_{dx} = n_{D_s}$	spatial derivative filter size
$N_{ax} = N_{ay} = n_{A_s}$	spatial averaging filter size
$N_{dt} = N_{dtt} = n_{D_t}$	temporal derivative filter size
$N_{at} = n_{A_t}$	temporal averaging filter size

Algorithm 2 Parallel Flux Tensor: SPE i

Input : Images in FIFO1, N_m, starting col C_i, and work block width W.
Output : Blocks of Intermediate Flux into FIFO2 and flux trace Tr_J_F

```
 1: for each row $r$ of Work Block do
 2:   Load from FIFO1, pixel data $I_r$ from column $C_i - \lfloor n_{D_s}/2 \rfloor$ up to $C_i + W + \lfloor n_{D_s}/2 \rfloor$ into local store
 3:   Push($I_r \otimes S_x, I_{S_x}\_buffer$);
      Push($I_r \otimes D_x, I_{D_x}\_buffer$);
 4:   if $I_{S_x}\_buffer$ and $I_{D_x}\_buffer$ have $n_{D_s}$ rows then
 5:     $I_{S_x D_y} = I_{S_x} \otimes D_y$;
 6:     $I_{S_x S_y} = I_{S_x} \otimes S_y$;
        $I_{D_x S_y} = I_{D_x} \otimes S_y$;
 7:     $I_{yt} = I_{S_x D_y} \otimes D_t$;
        $I_{tt} = I_{S_x S_y} \otimes D_{tt}$;
        $I_{xt} = I_{D_x S_y} \otimes D_t$;
        $F_r = I_{xt}^2 + I_{yt}^2 + I_{tt}^2$ ;
 8:     Push($F_r$, FIFO2);
        Pop($I_{S_x}\_buffer$);
        Pop($I_{D_x}\_buffer$);
 9:   end if
10: end for
11: if $N_m \geq n_{A_t}$ then
12:   for each row $r$ of Work Block do
13:     Load from FIFO2, $F_r$ data from column $C_i - \lfloor n_{A_s}/2 \rfloor$ up to $C_i + W + \lfloor n_{A_s}/2 \rfloor$ into local store
14:     Push($F_r \otimes A_x$,$I_{A_x}\_buffer$);
15:     if $I_{A_x}\_buffer$ has $n_{A_s}$ rows then
16:       $I_{A_x A_y} = I_{A_x} \otimes A_y$;
17:       $Tr\_J_F = I_{A_x A_y} \otimes A_t$;
          Pop($I_{A_x}\_buffer$);
18:     end if
19:   end for
20: end if
```

Black River System's notation in excel file:

$$\begin{aligned} f_x = f_y =& n_{D_s} \\ I_x = I_y =& n_{A_s} \\ f_t =& n_{D_t} \\ I_t =& n_{A_t} \end{aligned}$$

Sequential Flux Tensor Computation

This computation accounts for both multiplications and additions. The total is given per frame for a $P \times Q$ image.

$$\begin{aligned} I_{xs} =& P(2(Q - n_{D_s} + 1)n_{D_s} - 1) + Q(2(P - n_{D_s} + 1)n_{D_s} - 1) \\ I_{sy} =& P(2(Q - n_{D_s} + 1)n_{D_s} - 1) + Q(2(P - n_{D_s} + 1)n_{D_s} - 1) \\ I_{ss} =& P(2(Q - n_{D_s} + 1)n_{D_s} - 1) + Q(2(P - n_{D_s} + 1)n_{D_s} - 1) \\ I_{xst} =& PQn_{D_t} \\ I_{syt} =& PQn_{D_t} \\ I_{sst} =& PQn_{D_t} \\ \text{Flux}_{\text{trace}} =& 5PQ \\ \text{Avg}_{xy} =& P(2(Q - n_{A_s} + 1)n_{A_s} - 1) + Q(2(P - n_{A_s} + 1)n_{A_s} - 1) \\ \text{Avg}_{xyt} =& PQn_{A_t} \\ \text{Total}_{\text{frame}} =& I_{xs} + I_{sy} + I_{ss} + I_{xst} + I_{syt} + I_{sst} + \text{Flux}_{\text{trace}} + \text{Avg}_{xy} + \text{Avg}_{xyt} \\ \text{Total}_{\text{frame}} =& 2PQ(6n_{D_s} + 3n_{D_t} + 2n_{A_s} + n_{A_t} + 5) \\ & + (P + Q)(6n_{D_s}(1 - n_{D_s}) - 3 + 2n_{A_s}(1 - n_{A_s}) - 1) \\ \text{Mflop} =& \text{Total}_{\text{frame}}/1e6 \end{aligned}$$

Parallel Flux Tensor Computation (Black River System's estimate)

This computation accounts for both multiplications and additions. The total is given per frame. W is work unit width.

$$\begin{aligned} I_{dx} =& W(2n_{D_s} - 1)n_{D_t} \\ I_{sx} =& W(2n_{D_s} - 1)n_{D_t} \\ I_{sysx} =& W(2n_{D_s} - 1)n_{D_t} \\ I_{dysx} =& W(2n_{D_s} - 1)n_{D_t} \\ I_{sydx} =& W(2n_{D_s} - 1)n_{D_t} \\ I_{xt} =& W(2n_{D_t} - 1) \\ I_{yt} =& W(2n_{D_t} - 1) \\ I_{tt} =& W(2n_{D_t} - 1) \\ \text{Avg}_{x} =& W(2n_{A_s} - 1)n_{A_t} \\ \text{Avg}_{xy} =& W(2n_{A_s} - 1)n_{A_t} \\ \text{Avg}_{xyt} =& W(2n_{A_t} - 1) \\ \text{Flux}_{\text{out}} =& 5W \end{aligned}$$

$$\begin{aligned}
\text{Total} =& I_{dx} + I_{sx} + I_{sysx} + I_{dysx} + I_{sydx} + I_{xt} + I_{yt} + I_{tt} + \text{Avg}_x + \text{Avg}_{xy} + \text{Avg}_{xyt} + \text{Flux}_{\text{out}} \\
\text{Total} =& 5W(2n_{D_s} - 1)n_{D_t} + 3W(2n_{D_t} - 1) + 2W(2n_{A_s} - 1)n_{A_t} + W(2n_{A_t} - 1) + 5W \\
\text{Total}_{\text{frame}} =& \text{Total} \times \text{height} \\
\text{Mflop} =& \text{Total}_{\text{frame}} \times (\#\text{SPEs})/1e6
\end{aligned}$$

3.5.6 New Implementation of Flux Tensor

The flux tensor computation can be reorganized by exploiting the commutative property of convolution to change the order of the spatial and temporal derivatives as shown in Figure 13. This decreases the memory footprint as well as the operation count. This enables the new implementation to operate on complete lines of the image instead of just a (vertical) slice. The resulting multicore SPE code is simpler. But there is a higher communication overhead relative to the operation count. This means that the new flux tensor algorithm is less efficient in the use of the Cell/B.E. processor but is much faster. The new flux tensor implementation is about four times faster (from about 740 MFlops to 209 MFlops) and is not computation limited but rather communication limited at this point. The performance characteristics and differences for the new flux tensor implementation compared to the previous implementation are still being studied.

3.6 Binary Mask Processing Using Morphology Operators

Morphological operations process an input image by applying a structuring element and producing an output image where each pixel is based on a comparison of the corresponding pixel in the input image with its neighbors depending upon the size and shape of the structuring element. The most basic morphological operations are dilation and erosion and all others like opening, closing etc. consist of some combination of these two operations. Dilation, denoted by the operator $\oplus$, adds pixels to the boundaries of objects in an image, while erosion, denoted by the operator $\ominus$, removes pixels on object boundaries. Mathematically, these operations on binary images can be defined using set theoretic notations as:

$$A \oplus B = \{z | (\hat{B})_z \cap A \neq \emptyset\} \tag{21}$$

$$A \ominus B = \{z | (B)_z \subseteq A\} \tag{22}$$

where $\hat{B}$ is the reflection of set B and $(B)_z$ is the translation of set B by point z as per the set theoretic definition. So in dilation, if any pair of pixels in A and B are one is set to the value 1, the output pixel is set to 1. In erosion, if any pair of pixels in A and B-complement is set to 0, the output pixel is set to 0. In a recent work,[41] the authors describe their parallel implementation of morphology on Cell/B.E. for OpenCV interface. However they use one byte to represent pixel data and hence can process only 16 pixels simultaneously by utilizing the 128-bit SIMD computing unit on each SPE. Since in blob segmentation algorithm, these operations are applied on a binary mask, our implementation represents each pixel by a single bit and utilizes bitwise comparison AND/OR SIMD operations to process 128 pixels simultaneously. The input image can be processed using square, circular and elliptical shaped structuring elements depending on the type of image blobs to be processed. Each row of structuring element is represented using one byte element with appropriate bit pattern stored in a array SE_values and which is replicated to 16 byte vector array SE_Mask to operate on 128 pixel data of I at once. However, this required appropriate bit manipulation strategy for pixel based data access (i.e., bit by bit) as required by the algorithm. Hence, specialized shuffle (*spu_shuffle*) and bit rotate operations (*spu_rlqw*) are needed to access pixel data that is located on arbitrary alignment or the boundary of the 16 byte alignment. The *spu_shuffle* combines the bytes of two vectors as per an organization pattern defined in third vector. The required alignment patterns are stored in static look-up tables in the SPU local store. Figures showing how to access a 16 byte aligned vector data (*current*) and its neighbor vector data with one byte shift in left (*previous*) and right (*next*) direction, showing how to get further bit by bit access by bit rotate and shuffle operations and the subsequent comparison operations and showing the subsequent comparison operations can be found in our publication.[23]

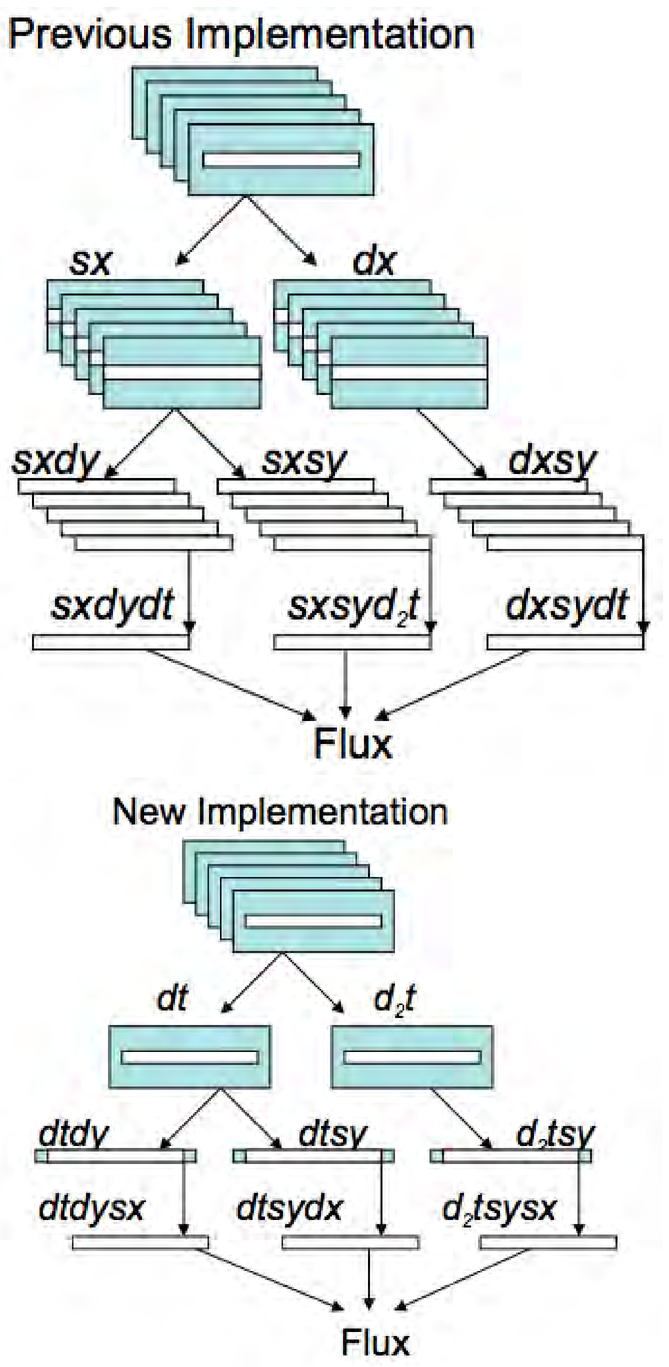


Figure 13: New parallelization approach for flux tensor is about four times faster than the previous version.

3.7 Blob Extraction by Connected Component Labeling

Connected Component Labeling (CCL) scans an image and groups its pixels into components based on pixel connectivity. In the first step, the image is scanned pixel-by-pixel (from top to bottom and left to right) in order to identify connected pixel regions, i.e. regions of adjacent pixels which share the same set of intensity values and temporary labels are assigned. CCL works on binary or graylevel images and different measures of connectivity (4-connectivity, 8-connectivity etc.) are possible. For our blob segmentation, the input is binary image and 8-connectivity measure is considered. After completing the scan, the equivalent label pairs are sorted into equivalence classes and a unique label is assigned to each class. As a final step, a second scan is made through the image, during which each label is replaced by the label assigned to its equivalence classes.

The proposed parallelization approach for the Cell/B.E. heterogeneous architecture belongs to the class of divide and conquer algorithm where the PPE runs the main process, divides the image into multiple regions or data chunks for allocating the task of labeling to multiple threads running on the various SPEs and finally merges the results from different SPEs to generate actual label within the entire image. Each SPE performs DMA data loading from the system memory and labeling the allocated region in an independent way. As argued previously, division of data into blocks of rows is preferred because a bulk of data (several rows) can be transferred per DMA request issue. Some components may span over multiple regions and to ensure that such components get unique label during merging, each region is divided in such a manner that it has one overlapping row with its adjacent regions. Figure 14 shows the main phases of our parallelization approach.

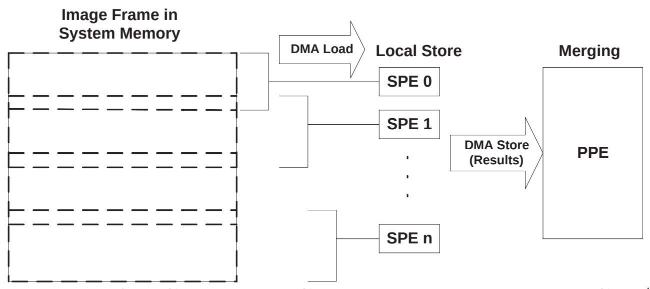


Figure 14: Parallelization model for Connected Component Labeling on Cell/B.E. showing adjacent tile overlap or hook regions.

The resultant array of local labels within each region is put back into main memory through DMA store operation. The PPE uses a list of pointers $Region[i]$ to point to arrays that maintain the local labels with respect to SPE_i. Initially, index for each array element is the local label before equivalence resolution whereas the array element itself is the local label after equivalence resolution in the corresponding regions. Since the labeling starts from value 1, the array location for index 0 is used to store the total number of distinct labels after resolution of equivalence class of labels. To connect each region with its neighboring regions to generate the actual label within the entire image, the PPE updates the array elements in $Region[i]$ by adding the total labels T that reached at the end of $Region[i-1]$. Figure 15 depicts an example of list of labels for $Region[i]$ which shows that local label 1, 2 and 5 are equivalent with local label 1 and their global label within the entire image is $T+1$; local label 3, 4, and 6 are equivalent with local label 2 and their global label is $T+2$ where T is the total labels reached at the end of $Region[i-1]$. While updating the labels in the global list we also need to resolve equivalences of pixel labels for the overlapping row between regions. To do this we use a list of pointers, $OverlapBottom[i]$ and $OverlapTop[i]$ to store the local labels for the overlapping pixel in the bottom and the top row respectively, of the region processed by SPE_i. Now for any pixel k in the overlapping row between regions i and $i-1$, the local labels as calculated by SPE_i and SPE_{i-1} are stored in $OverlapTop[i][k]$ and $OverlapBottom[i-1][k]$ which should be equivalent. We use this information to resolve equivalence of labels across different regions for merging.

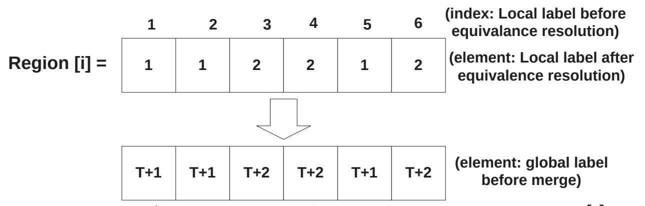

1
2
3
4
5
6
(index: Local label before equivalance resolution)
Region [i] =
1
1
2
2
1
2
(element: Local label after equivalence resolution)
T+1
T+1
T+2
T+2
T+1
T+2
(element: global label before merge)

Figure 15: An example of list of labels in Region[i]

The part of the proposed algorithm implemented on the PPE side is presented in algorithm 3. In the beginning, four buffers are created: $Frame, Region, OverlapTop$ and $OverlapBottom$. These buffers are used to store the frame pixels to be sent to the SPEs and the results obtained from each SPE. The image is divided into N (the number of available SPEs) regions or data blocks with an overlap of one row at the top and bottom of each image region with the adjacent regions. The first region and the last region do not have any overlap at the top and bottom respectively. Correspondingly, the address locations in the buffers are determined and send as control block to the SPE thread. The SPE performs DMA operations to load the input data and store the results at these locations. PPE waits for the SPE to finish computing and notify the total number of local labels through mailbox communication. Then it updates the labels by going through each $Region[i]$ one by one, adding the total labels reached till previous region and resolving the equivalence by calling merge function.

Algorithm 3 Parallel Connected Component Labeling: PPE Side

```
1: Allocate $Frame, Region, OverlapTop, OverlapBottom$ and control block $cb$ buffer
2: Divide image into N regions along with the overlaps
3: for each SPE_thread $i$ do
4:     Create thread $i$ to run on SPE $i$
5:     Load the control blocks $cb[i]$ with the corresponding address location for
       $Frame, Region, OverlapTop, OverlapBottom$ buffer to the thread
6: end for
7: for each SPE_thread $i$ do
8:     Get Total number of local labels from SPE $i$ mailbox into $Labels[i]$
9:     Wait for thread joining and destroy the context
10: end for
11: Initialize total number of labels $T \leftarrow 0$
12: for each region $i$ do
13:     for each label index $j = 1$ to $Labels[i]$ do
14:         $Region[i][j] \leftarrow T + Region[i][j]$
15:     end for
16:     $T \leftarrow T + Region[i][0]$
17:     if region is not first then
18:         $Merge(i, T, Region, OverlapTop, OverlapBottom)$ function to update global labels and value of $T$
19:     end if
20: end for
```

On the SPE side, the first stage of the algorithm is essentially like any standard sequential algorithm which scans the allocated region of the image pixel by pixel, assigning a temporary label to a new pixel and marks the labels of connected pixels as equivalent. However, the algorithm used to resolve the equivalence class is implemented in a way that utilizes the SIMD instructions. The *Union-Find* algorithm which efficiently resolves the equivalence class of labels for sequential execution was not preferred because of its search based flow to keep the height of the search tree minimum. SPEs deliver most of the computational capacity by issuing vectorized instructions in SIMD fashion. Thus, the choice of an algorithm for implementing an application on SPE depends more on how the operations can be grouped for issuing in SIMD fashion to utilize the 128-bit SIMD units which can operate on 16 8-bit integers, eight 16-bit integers, four 32-bit integers, or four single precision floating-point numbers in a single clock cycle. Hence, we choose to implement an alternative algorithm that resolves equivalences by expressing equivalent relations as a binary matrix and applying Floyd-Warshall algorithm to obtain transitive closure. An interesting point to note about this algorithm is that although it takes $O(n^3)$ OR operations, these operations can be calculated in parallel very efficiently on the SPE using SIMD bitwise OR operations in hardware reducing approximately 16 OR operations to a single (clock-cycle) operation. The corresponding vectorized code on the SPE side is as follows:

Listing 2: SPE Code Segment for Equivalence Class Resolution Using Vectorized SIMD bitwise OR Operations

```
for (j=1;j<n;j++){
  for (i=1;i<n;i++){
    if (T[i*n+j]==1){
      vec_uchar16* v1=(vec_uchar16*) & T[i*n];
      vect_uchar16* v2=(vect_uchar16 *) & T[j*n];
      for (k=1;k<size/16;k++)
        v1[k]=spu_or(v1[k],v2[k]);
} } }
```

Due to the memory constraint on the local store, only a part of the image data can be brought into the SPE local store at any given time. This required the implementation to handle the spatial dependency on the previous row pixel, whenever a new block of rows is fetched. This was done by keeping two arrays *OverlapTop* and *OverlapBottom* that store the labels of the pixels in the top and bottom row of the present block of rows. The bottom row for the previous block of rows is used for checking pixel connectivity for the first row of the next block of rows and the *OverlapBottom* is updated for the new block of row. Finally when the scan is complete, *OverlapTop* and *OverlapBottom* array is send to the PPE for use in merging labels in the adjacent regions. The SPE also sends out the array of labels after resolving the equivalences of classes, in the buffer *Region*[*i*].

4. RESULTS AND DISCUSSION

4.1 Multicore Flux Tensor Performance

The output of the flux tensor-based video object detection algorithm applied to a sample video sequence from the ARL Force Protection Surveillance System (FPSS) video collection[11] is shown in Figure 16. The first row shows color and long-wave infrared frames from the original video sequence. The second and third rows show the grayscale flux tensor response and the thresholded binary masks respectively using the flux tensor motion analysis with $(5, 5, 5, 5)$ filters, followed by grayscale closing (circular structuring element of radius 5) and using histogram based thresholding, adaptively switching between global Otsu and 80% cumulative histogram value. The colored blobs show the detected moving objects after post processing steps including morphological noise removal using area opening and connected component labeling to identify contiguous regions. The pink blobs are associated with two people walking in the far background. The FLIR channel is not affected by shadows and produces more compact blobs of moving objects suitable for tracking.

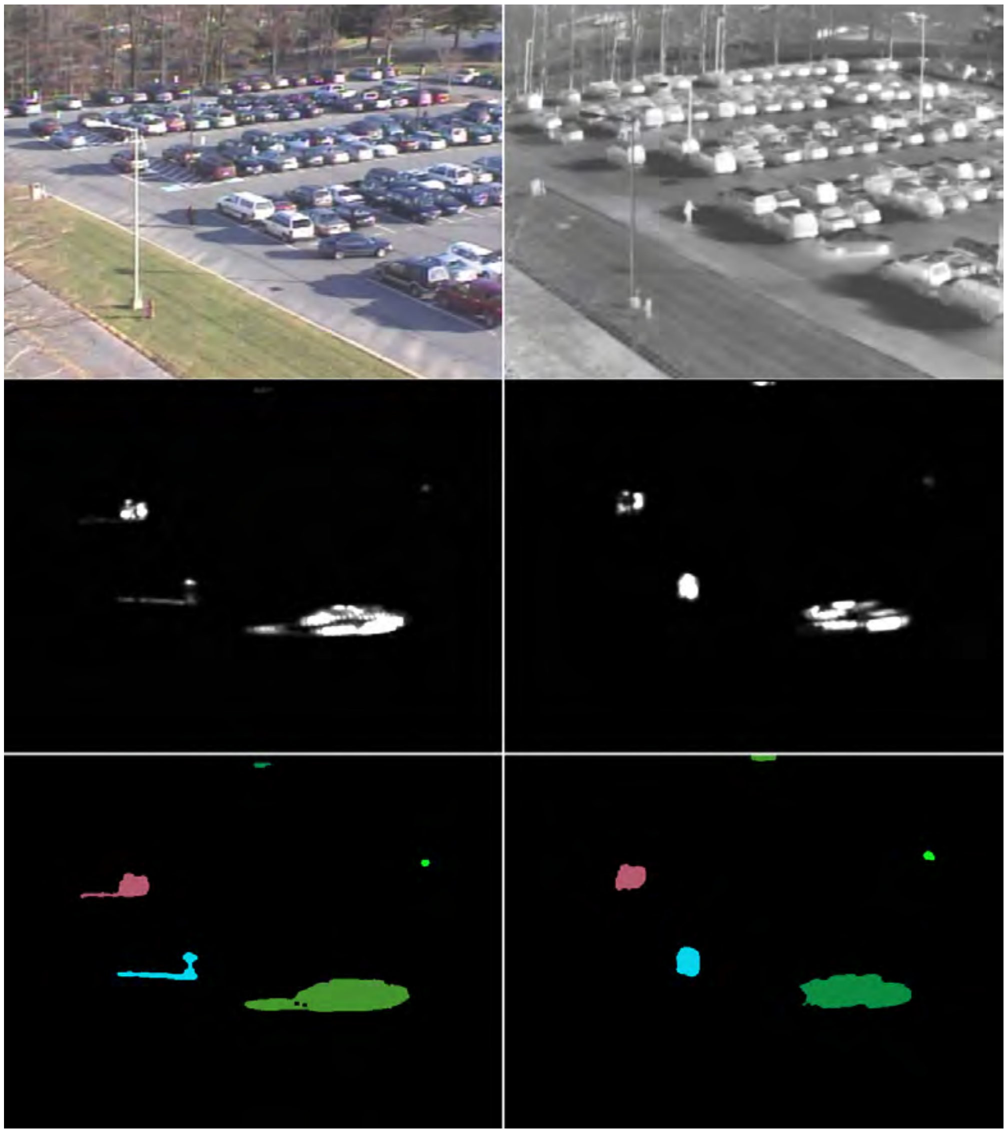


Figure 16: Output of flux tensor motion estimation and blob extraction algorithm on selected frames of color visible and FLIR (forward looking long-wave infrared) data from the ARL FPSS dataset.[11]

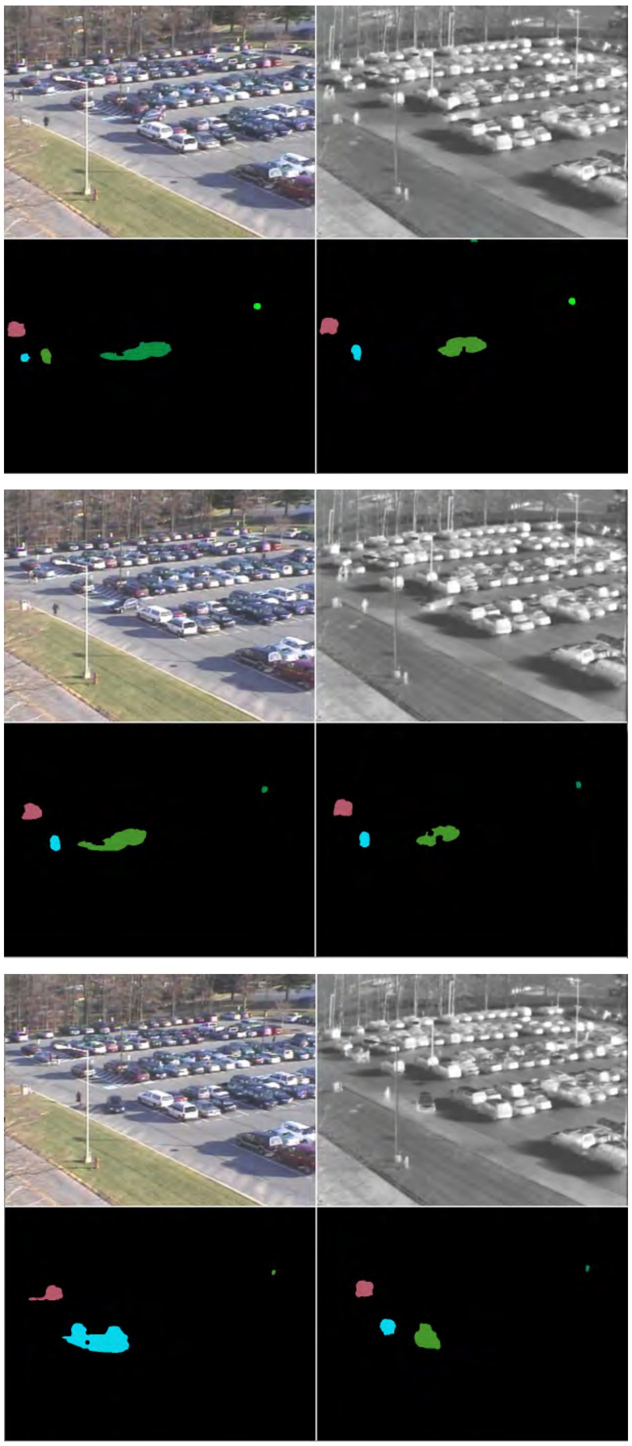


Figure 17: Results for several intermediate frames in this long FPSS sequence.

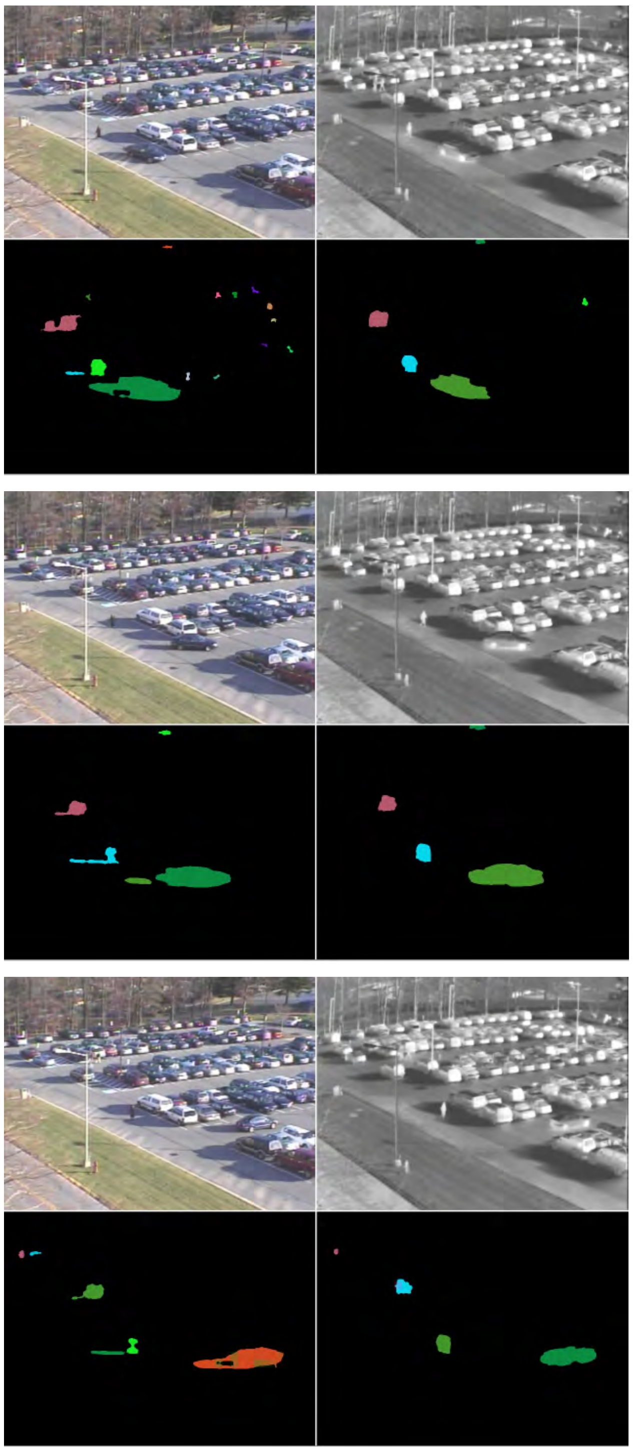


Figure 18: Results for three additional intermediate frames in this long FPSS sequence.

Table 1: Speedup and performance to power ratios of the parallel multicore PS-3 Cell/B.E. implementation with 6 SPEs using 135 watts is compared to the optimized sequential implementation running on an Intel Xeon core in a Dell 1850 server using 550 watts. Sequential performance in frames per second are shown in the $(T_1^{\text{Seq}})^{-1}$ columns. Parallel PS-3 speed-up $(S_6^{\text{PS-3}})$ is compared to the sequential implementation running on an Intel Xeon CPU. The power efficiency improvements of the parallel implementations compared to the sequential implementations for different filter sizes are shown in the *PPR* columns.

Filter Configuration				**HD video**			**SD video**		
n_{D_s}	n_{D_t}	n_{A_s}	n_{A_t}	$(T_1^{\text{Seq}})^{-1}$	$S_6^{\text{PS-3}}$	PPR	$(T_1^{\text{Seq}})^{-1}$	$S_6^{\text{PS-3}}$	PPR
3	3	3	3	1.75	40.1	164	17.32	18.2	74
5	3	5	3	1.68	38.6	157	14.35	20.0	82
7	3	7	3	1.54	38.1	155	13.02	19.3	79
9	3	9	3	1.37	39.6	161	11.52	19.6	80
3	5	3	5	1.53	30.8	125	13.99	14.7	60
5	5	5	5	1.42	29.6	120	12.28	14.9	61
7	5	7	5	1.34	28.7	117	10.99	15.2	62
9	5	9	5	1.23	28.3	115	10.05	14.8	60
3	7	3	7	1.34	25.6	104	12.01	13.5	55
5	7	5	7	1.25	24.7	101	10.44	13.2	54
7	7	7	7	1.18	23.6	96	9.88	11.9	48
9	**7**	**9**	**7**	**1.11**	**14.2**	**58**	9.03	12.0	49
3	9	3	9	1.24	22.0	89	10.45	12.0	49
5	9	5	9	1.14	21.3	87	9.34	11.3	46
7	**9**	**7**	**9**	**1.09**	**12.4**	**51**	8.63	11.1	45
9	**9**	**9**	**9**	**1.02**	**13.0**	**53**	7.97	10.7	43

The sequential code was tested on a Dell PowerEdge 1850 server running CentOS Linux 5.4 using a single core of a dual CPU dual core Intel Xeon 2.8 GHz with 2 MB of cache per core, 4GB of memory and an 800MHz front side bus compiled using gcc -O3 version 4.1.2. The parallel code was tested on a PS3 Cell/B.E. with 6 SPEs using an appropriate SPE work unit for HD sized (1920 × 1080, WU=320 or smaller) and Standard Definition (SD) sized (640 × 480, WU=112 pixels) images. The PS-3 uses about 135 watts while the Dell PowerEdge 1850 uses about 550 watts for system operation including CPU, peripheral devices, operating system, multitasking, etc. Total system power was used to measure the performance to power efficiency ratios without doing detailed power measurements that can become complex to instrument and compare. The work unit size that can be accommodated by one SPE with 256 KB of local store depends on the size of the 3D convolution filters, especially the temporal filters, data alignment and partitioning requirements (usually multiples of 16 bytes). Parallel performance benchmarking done on an IBM QS20 and QS22 Blade servers with dual Cell/B.E.s both running Fedora Linux all compiled using gcc -O3 version 4.1.2 are reported below.

We compared the performance between the sequential and parallel implementations of flux tensor for different filter configurations on two different frame sizes of video streams using 3D grids. Speed-up using p processors was calculated as,

$$S_p^{\text{PS-3}} = \frac{T_1^{\text{Seq}}}{T_p^{\text{PS-3}}} \tag{23}$$

where T_p is the average time measured across p processors to complete the flux tensor computation for one frame and T_1^{Seq} is the time taken for the single core sequential implementation; $p = 6$ SPEs on the PS-3. The performance to power efficiency improvement ratio was calculated as,

$$PPR = S_p^{\text{PS-3}} \frac{P_{\text{Seq}}}{P_{\text{PS-3}}} \tag{24}$$

where P_A is the system power used by architecture A and $S_p^{\text{PS-3}}$ is the speed-up ratio.

Table 1 shows the range of spatial and temporal sizes for both derivative and integral/averaging filters varying between $3, 5, 7$ and 9 that were used for performance benchmarking. The sequential frame rate or inverse of the time to compute one frame on the Intel Xeon processor is given in column $(T_1^{Seq})^{-1}$ in frames per second, the speed-up measured on the PS-3 Cell/B.E. platform compared to the sequential performance is given in column $S_6^{\text{PS-3}}$. For the smallest derivative and integral filters of size 3 the speed-up of the parallel implementation compared to the sequential performance was more than a factor of 40 even though there are only six computational cores. The super-linear speed-up behavior is due to the use of extensive vectorization, loop unrolling and FMA operations to implement the flux tensor convolution kernels despite the additional work done by the SPEs recomputing intermediate results in the parallel implementation compared to the sequential version. As the filter sizes increase the speed-up gain decreases since the total volume of computations increases faster on the Cell (linearly with the size of the filter) as the parallel implementation recomputes all of the intermediate spatial derivatives, whereas the sequential implementation is able to store and reuse them at the cost of extra memory. For larger filter sizes the limited local store on the SPEs requires smaller data chunks (work unit width in Figure 11) and consequently more than six threads of execution which results in two stages of computation and reduces speed-up; the filter configurations needing two stages of execution are shown in bold font in Table 1. This can be partly mitigated by using the more expensive IBM Cell/B.E. Blade processors which have a lot more main memory but also higher power consumption. More detailed performance figures for HD and SD video with work width (WW) size in pixels are given in Tables 2 and 3 respectively including speedups relative to the PS3, QS20 and QS22 Blades.

The speed-up of the multicore flux tensor implementation ranged from a factor of 11 to 20 for the smaller SD video frame sizes to between 12 and 40 for the larger HD frame size video streams as shown in Table 1. The results for 16 different filter configurations for both HD and SD video frame sizes are compared and show substantial improvement in terms of both parallel speed-up as well as performance to power efficiency. Our implementation on the PS-3 Cell/B.E. was able to deliver 34.9 fr/s $\times 1.32$ GFLOPs/frame = 46 GFLOP/s which is 30% of single-precision peak performance (153.6 GFLOP/s) but significantly better than the expected memory-intensive peak of 12.8 GFLOP/s. The identical code reaches 39%, 35% and 24% of peak performance on the QS20 (410 GFLOP/s for 16 SPEs) with 6, 8 and 16 SPEs respectively for the same filter size configuration of $n_{D_s} = 9, n_{D_t} = 5, n_{A_s} = 9, n_{A_t} = 5$. An earlier implementation that used better data alignments but could only handle a limited number of filter sizes and image widths was able to reach 68 fr/s and 58% of peak on the QS20. We found that explicit memory management and some assembly coding on the Cell is required to reach high performance even though this hand tuning incurs additional programming effort. Multicore GPU architectures are also well suited for computer vision algorithms.[17] In future work we will compare the energy efficiency of the flux tensor on GPUs based on a Compute Unified Device Architecture (CUDA) or OpenCL implementation.

Some specific performance results measured by Black River Systems to compare with the graphs produced with the assistance of Barcelona Supercomputing Center are given below. The BRS numbers measure only the time spent on the SPEs, but the BSC numbers usually include the SPE thread creation and destruction times also.

- 6 SPEs 1920x1080 5,5,5,5 0.016033 per frame 62 fps
- 6 SPEs 1920x1080 7,5,7,5 0.018675 per frame 53 fps
- 6 SPEs 640x480 5,5,5,5 0.004513 per frame 221 fps
- 6 SPEs 640x480 7,5,7,5 0.005353 per frame 186 fps

The set of graphs described in this section were based on an extensive number of experiments organized as follows: 100 frames per sequence, 9 filter sizes, 10 runs or repetitions per filter size, 16 different SPE configurations, two video formats. This requires a total of $9 \times 10 \times 16 \times 2 = 2880$ runs for mean and variance estimation for each data point which at 15 seconds per run took about 12 hours on the Cell Blade. For the HD size images testing sequential performance took about one minute per sequence and 15 seconds per run for SD video or about 30 hours total for the full experiment to get the comparison statistics in the tables.

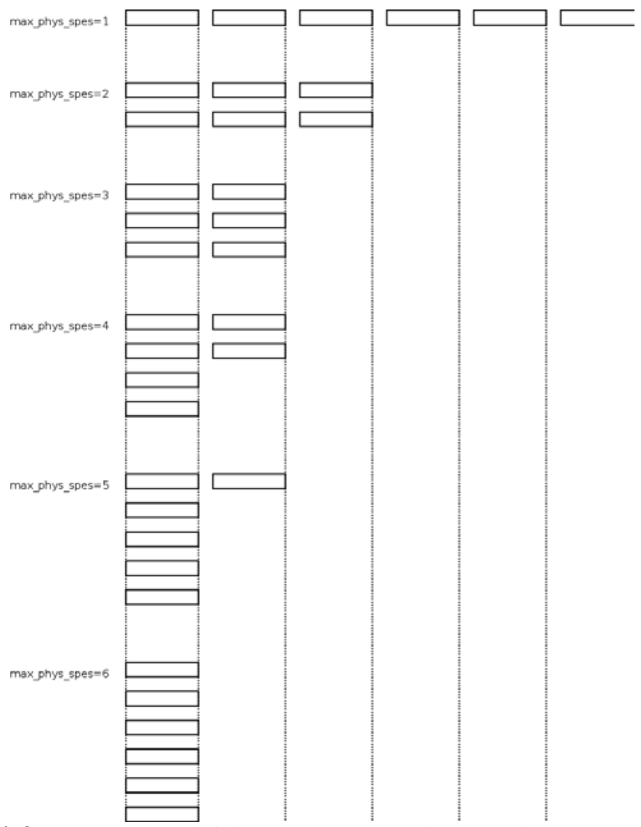


Figure 19: Dependence of flux tensor scalability on image tiling or slicing. With 4 and 5 SPEs there is no advantage over 3 SPEs since the chunking requires two stages of computation (in all three cases) to complete the calculation for the full frame. In these cases there are many idle SPEs (two and four idle cores respectively).

The chunking (or tiling or slicing) of the image into vertical or horizontal strips leads to certain constraints in the interpretation of scalability with increasing number of SPEs. In order to test the code for different numbers of SPEs the variable max_phys_spes in the source code is appropriately modified. The flux tensor application calculates the number of slices per image as num_spes. In Figure 20 the num_spes (tasks) is plotted against max_phys_spes for a particular filter configuration. For example, with 1 to 6 SPEs, each image is subdivided into 6 blocks. So the flux application for these parameters should scale well up to 3 SPEs, and not scale at all from 4 to 5 SPEs, etc. as shown in Figure 19. The large influence of the cost of thread creation and destruction can hide this dependence as shown in Figure 21. Here the flux tensor seems to scale well up to three SPEs when the SPE thread creation and destruction times are included. But when we ignore thread creation and destruction times then we get Figure 22 and Figure 23 for HD image sizes. These measurements do exhibit the flatlining behavior one can expect from Figure 20.

The proposed implementation can be used to serve any large scale surveillance system that requires scalable processing up to HD Wide-Area Motion Imagery. As a case study, the flux tensor parallelization required most of the significant effort (about 3 staff months) centered on efficiently mapping memory management and data alignment, data (DMA) transfers, vectorization and shuffling issues to partition the 3D grid of video data and fit each working block within the memory constraints on an SPE. Since the SPU has only 256 KBytes of memory and limited programmer development tools, this required manual code partitioning and coordinating data movement into and out of the SPE registers. For optimizing the major bulk of the flux tensor computation involving separable 3D convolutions on the SPE, we utilized fused vector multiply adds (FMA) to approach the peak SPU performance. However, the fused vector multiply add operation has a latency of 6 cycles. Thus, loop unrolling was essential and the number of unrolls was further dependent on

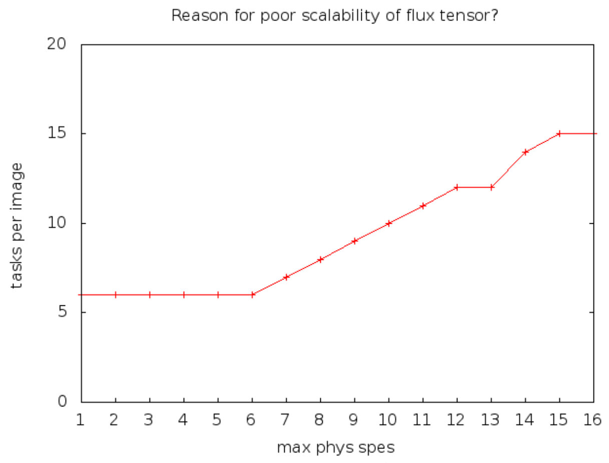


Figure 20: Number of tasks launched per image during performance testing with varying number of SPEs.

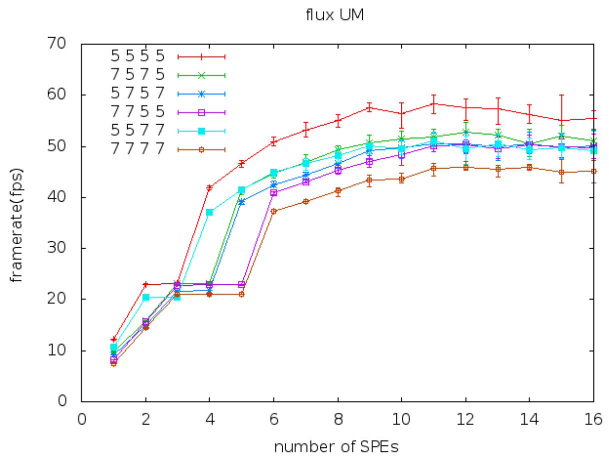


Figure 21: Including the cost of thread creation and destruction reduces the influence of image chunking.

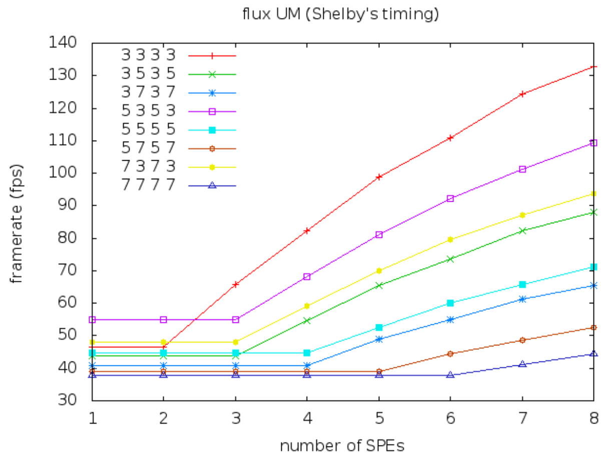


Figure 22: Excluding the cost of thread creation and destruction clearly shows the influence of image chunking by the flatline behavior that is dependent on filter size when the number of available SPEs is increased.

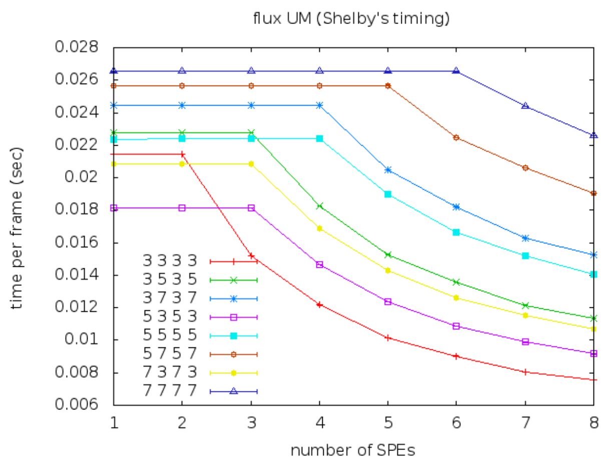
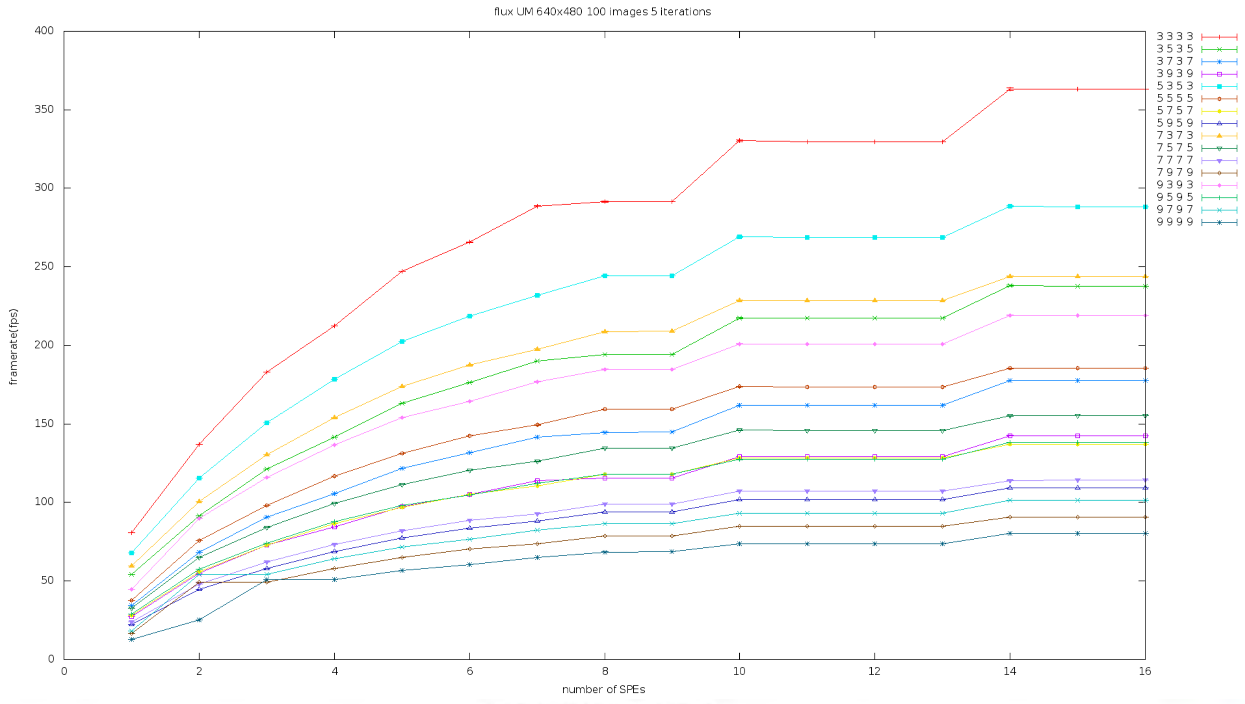


Figure 23: Performance plotted as time per frame (instead of fps) for the same data as in Figure 22.

Table 2: Parallel speed-up of the flux tensor operator implemented on the Cell/B.E. with 6 SPEs compared to the sequential implementation running on an Intel Xeon processor for nine different filter configurations and using two different sizes of 3D spatiotemporal grids ($1920 \times 1080 \times t$) for HD video. Typically up to ten repetitions were done for each test condition using 100 frames of video.

Filter Configuration				**HD video** ($1920 \times 1080 \times t$)							
n_{D_s}	n_{D_t}	n_{A_s}	n_{A_t}	WW	Seq fps	PS3 fps	PS3×	QS20 fps	QS20×	QS22 fps	QS22×
3	3	3	3	320	1.75	70.2	40.1	110.7	63.3	107.6	61.5
5	3	5	3	320	1.68	64.8	38.6	92.1	54.8	91.2	54.3
7	3	7	3	320	1.54	58.7	38.1	79.4	51.6	80.3	52.1
9	3	9	3	320	1.37	54.3	39.6	70.8	51.6	72.0	52.6
3	5	3	5	320	1.53	47.1	30.8	73.6	48.1	71.8	46.9
5	5	5	5	320	1.42	42.0	29.6	60.2	42.4	59.5	41.9
7	5	7	5	320	1.34	38.4	28.7	51.2	38.2	51.9	38.7
9	5	9	5	320	1.23	34.9	28.3	45.3	36.8	46.3	37.6
3	7	3	7	320	1.34	34.2	25.6	54.9	40.9	52.9	39.5
5	7	5	7	320	1.25	30.8	24.7	44.5	35.6	44.3	35.4
7	7	7	7	320	1.18	27.8	23.6	37.7	31.9	37.9	32.1
9	7	9	7	240	1.11	15.7	14.2	19.7	17.7	20.3	18.3
3	9	3	9	320	1.24	27.2	22.0	44.0	35.5	42.7	34.5
5	9	5	9	320	1.14	24.3	21.3	35.5	31.1	35.3	31.0
7	9	7	9	240	1.09	13.6	12.4	17.6	16.2	18.0	16.5
9	9	9	9	192	1.02	13.3	13.0	17.7	17.3	18.4	18.1

the work block width. An additional challenge was in moving the data from local store to the SPU registers. The Cell/B.E. processor is optimized for loading data from the local store that is aligned into 128-bit vector cache lines. Since the data necessary for the x-dimension convolutions overlap, those entries that are not aligned can be easily produced from the values that are aligned. SPU shuffle operations were used to create a result vector from two input vectors via selection of their constituent bytes which is more efficient than explicitly unpacking and repacking the vectorized data. Shuffle operations were not needed for convolutions in y- and t-directions since all the input vectors are separate and can be loaded as aligned vectors. The current implementation supports larger image frame sizes and variable filter sizes in an efficient manner as long as the partitioned 3D grid data and associated program instructions can fit within the limited SPE private memory (256 Kbytes local store). Once moving objects have been detected then Cell/B.E. based or GPU-based parallel algorithms for blob extraction using morphology operations and connected component



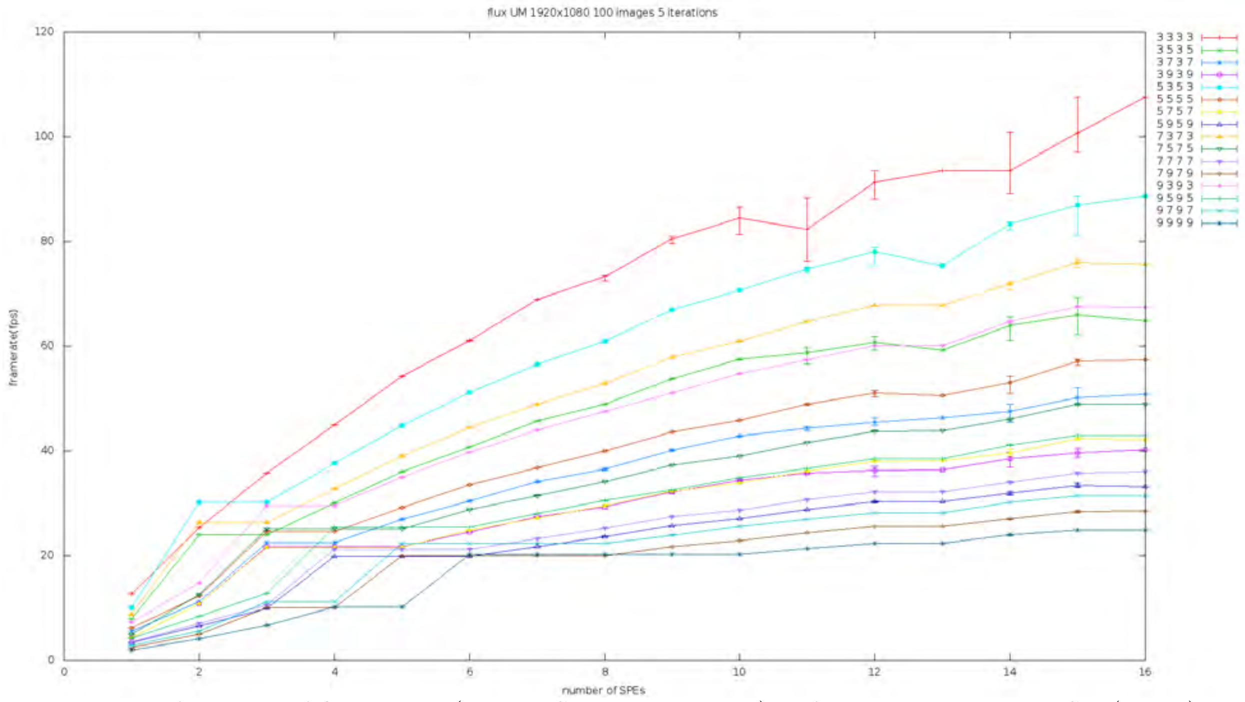


Figure 24: Performance of flux tensor (MU-BRS implementation) in frames per second on SD (upper) and HD (lower) video using different number of SPE cores on the IBM QS22 Blade to characterize scalability. Each plot is for a different filter size as shown in the legend to the upper right and tested on 100 images five repetitions each to get averages. The legend are derivative and averaging filter tap sizes a b c d to be interpreted as $fx_{tap} = a, fy_{tap} = a, ft_{tap} = b, ix_{tap} = c, iy_{tap} = c, it_{tap} = d$.

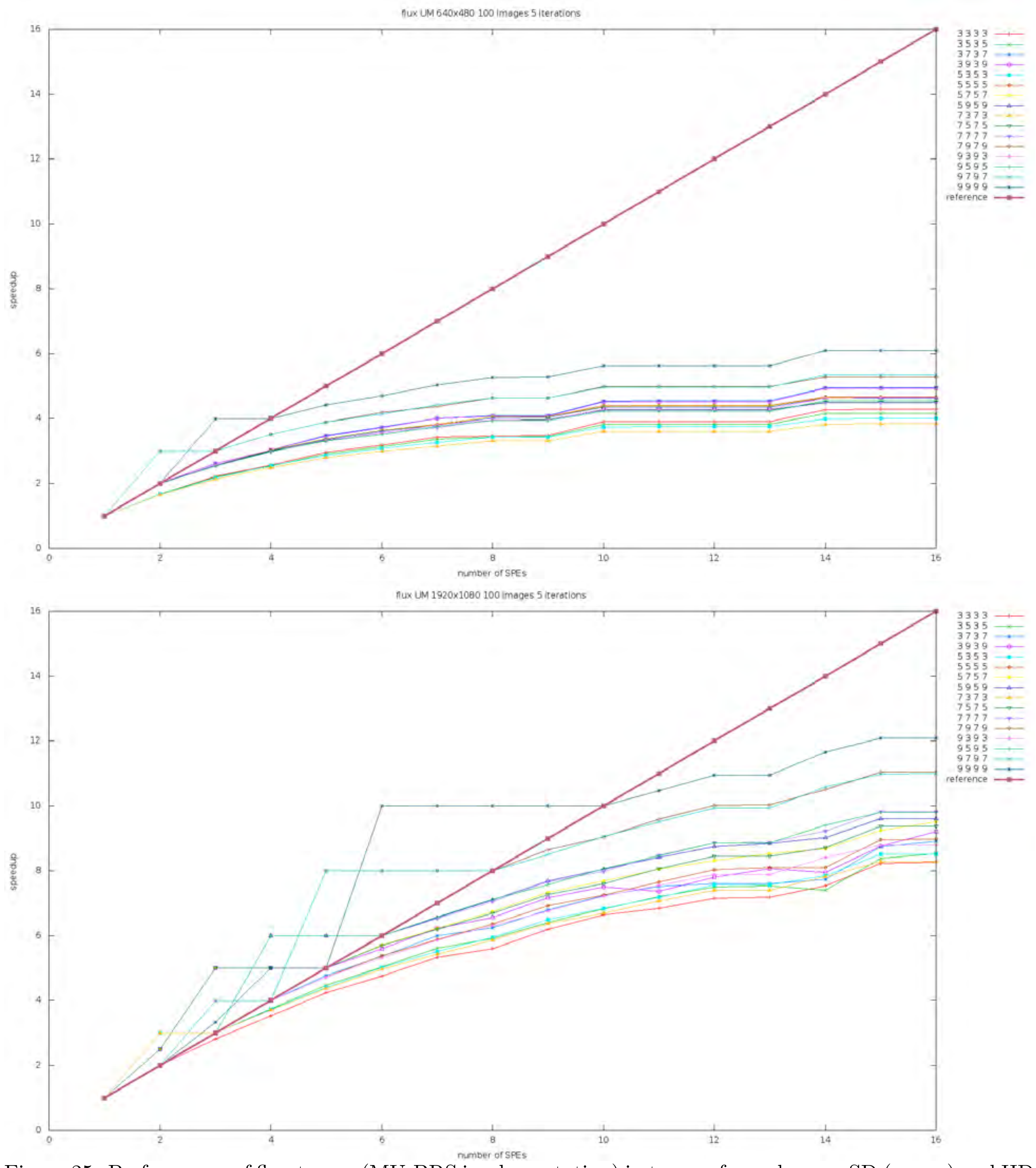


Figure 25: Performance of flux tensor (MU-BRS implementation) in terms of speed-up on SD (upper) and HD (lower) video compared to using one SPE cores on the IBM QS22 Blade to characterize scalability. Ideal linear speedup is shown in the solid red line. Each plot is for a different filter size as shown in the legend to the upper right and tested on 100 images five repetitions each to get averages. The legend are derivative and averaging filter tap sizes a b c d to be interpreted as $fx_{tap} = a, fy_{tap} = a, ft_{tap} = b, ix_{tap} = c, iy_{tap} = c, it_{tap} = d$.

Table 3: Parallel speed-up of the flux tensor operator implemented on the Cell/B.E. with 6 SPEs compared to the sequential implementation running on an Intel Xeon processor for nine different filter configurations for SD video ($640 \times 480 \times t$). Typically up to ten repetitions were done for each test condition using 100 frames of video.

Filter Configuration				**SD video** ($640 \times 480 \times t$)							
n_{D_s}	n_{D_t}	n_{A_s}	n_{A_t}	WW	Seq fps	PS3 fps	PS3×	QS20 fps	QS20×	QS22 fps	QS22×
3	3	3	3	112	17.32	315.0	18.2	473.0	27.3	481.9	27.8
5	3	5	3	112	14.35	287.4	20.0	385.2	26.8	399.2	27.8
7	3	7	3	112	13.02	251.6	19.3	326.4	25.1	343.3	26.4
9	3	9	3	112	11.52	225.6	19.6	286.3	24.9	301.2	26.1
3	5	3	5	112	13.99	205.5	14.7	311.9	22.3	318.5	22.8
5	5	5	5	112	12.28	183.1	14.9	249.3	20.3	257.1	20.9
7	5	7	5	112	10.99	167.5	15.2	208.7	19.0	220.6	20.1
9	5	9	5	112	10.05	148.9	14.8	181.6	18.1	192.0	19.1
3	7	3	7	112	12.01	162.3	13.5	232.2	19.3	236.2	19.7
5	7	5	7	112	10.44	137.5	13.2	184.0	17.6	192.2	18.4
7	7	7	7	112	9.88	117.6	11.9	153.3	15.5	161.6	16.4
9	7	9	7	112	9.03	108.4	12.0	132.9	14.7	141.0	15.6
3	9	3	9	112	10.45	125.3	12.0	185.3	17.7	189.5	18.1
5	9	5	9	112	9.34	105.5	11.3	146.1	15.6	152.2	16.3
7	9	7	9	112	8.63	95.6	11.1	121.2	14.0	128.0	14.8
9	9	9	9	112	7.97	84.9	10.7	104.9	13.2	111.8	14.0

labeling can be used.[23, 29] More extensive performance benchmarking and testing in the context of the overall end-to-end system for multi-stage video analysis is ongoing.

4.2 Multicore Morphology Performance

In our experiments with morphological processing, we compared the performance of our proposed parallel multicore implementation on the Cell/B.E. to the sequential version as well as a published OpenCV implementation for the Cell/B.E.[41] The experiment was performed using all 6 SPEs and varying the size and shape of the structuring element. The sequential code implemented in OpenCV was executed only on the PPE to measure the sequential performance and the proposed parallel implementation was executed on the SPEs to measure the parallel performance and speedup. Table 4 shows the measured execution time of one dilation or erosion operation averaged over 10 executions when the input image size is 1024×768. The original code implemented only on the PPE represents the sequential performance baseline and the optimized code implemented on the SPEs represents the parallel performance achieved on the Cell/B.E. As shown in Table 4, our implementation on the Cell/B.E. successfully achieved very high speed up ratios as high as 628 and significantly outperforming the published Cell/B.E. OpenCV benchmarks. The speedup increased for larger sized structuring elements i.e., as the amount of computation increased. Further interesting point to note is that for a given size of structuring element, regardless of the structuring element shape our implementation has roughly constant cost, whereas the sequential and OpenCV implementations become significantly slower as reflected in Table 4. This is because our representation of the structuring element uses the same number of bits for a given window size irrespective of shape which has a dramatic impact on the morphology computation for elliptical structuring elements. Figure 26 shows the performance measurements for morphology using three different structuring elements on SD video for one opening and one closing. Figure 27 measures speedup compared to a baseline of one SPE and performance for one opening and one closing. The measurements shown are for time spent in the CellSs tasks. Each graph plots results for a particular structuring element ("SE") and for a particular blocking including for example, the average and standard deviation for structuring element one where each task processes three rows per task. A blocking factor of 60 divides the image neatly into 8 panels, resulting in almost perfect speedup.

Table 4: Performance comparison of our parallel morphology to the sequential and Cell/B.E. OpenCV[41] implementation. The execution time measured is for one dilation or erosion operation on a 1024 × 768 sized image.

Structuring Element		**Sequential OpenCV execution on PPE (ms)**	**Proposed parallel execution on SPE**		**Cell/B.E. OpenCV execution on SPE**	
Size	Shape		Time (ms)	Speedup	Time (ms)	Speedup
7 x 7	Ellipse	89.806	0.143	628.7	0.973	92.4
5 x 5		47.740	0.104	459.0	0.562	84.9
3 x 3		16.324	0.064	254.7	0.308	52.9
7 x 7	Rectangle	24.235	0.135	179.5	0.337	71.9
5 x 5		18.782	0.098	191.6	0.289	64.9
3 x 3		14.341	0.062	231.3	0.276	51.9

In the process of testing and evaluation a likely/potential coding error in the morphology code was discovered. For example, the function erode() contains the code fragment:

Listing 3: Row column indexing problem in morphology implementation

```
__inline__
void erode(vector unsigned char *rows[],
           unsigned char mask[],
           unsigned int row_len, unsigned int size,
           vector unsigned char *result)
{
  unsigned int i,j;
  for (i = 0; i < row_len; i++) {
        vector unsigned char buf_0 = FF;
        vector unsigned char buf_1 = FF;
        vector unsigned char buf_2 = FF;
        vector unsigned char buf_3 = FF;
        vector unsigned char buf_4 = FF;
        vector unsigned char buf_5 = FF;
        vector unsigned char buf_6 = FF;
        vector unsigned char buf_7 = FF;
        for (j=0; j < size; j++) {
             vector unsigned char xprev =
             spu_shuffle(rows[j][i-1],rows[j][i],get_left_overlap);
        }
}
```

The last assignment accesses "rows[0][-1]" on the first iteration, for example. Now, this might not be a problem if one takes special care to arrange the data, and explicitly take care of such constructions in the code. In general this implementation would have problems with the borders of the image, or needs additional checks to deal with the spillover in the overlapping regions.

4.3 Multicore CCL Performance

In evaluating our implementation of the CCL algorithm, we compared performance between different sequential and parallel versions. We implemented a sequential version of the *Union-Find* algorithm (*sequential_UnionFind*) and a sequential version of the *Floyd-Warshall* algorithm (*sequential_FW*) for execution on the PPE. We implemented parallel versions of the connected component labeling algorithm with and without SIMD instructions for resolution of equivalence class labels, referred to as *parallel_CCL* and *parallel_CCL_SIMD* respectively that execute on both the PPE and all 6 SPEs. We measured the results of

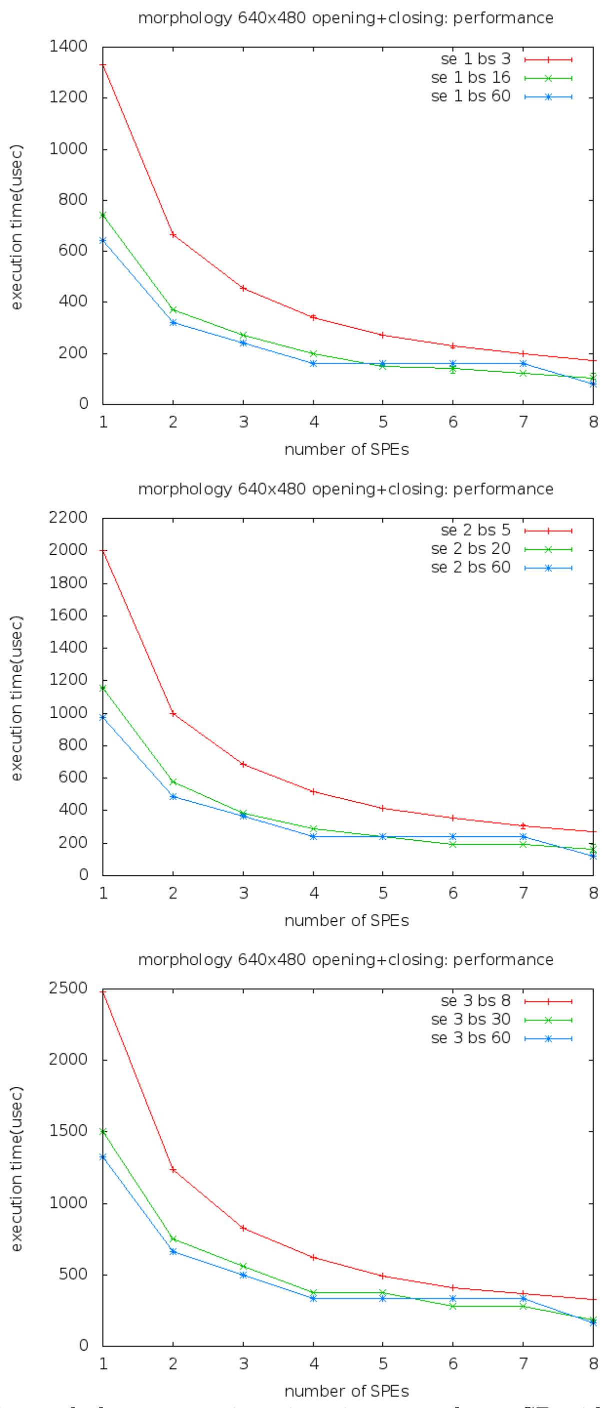


Figure 26: Performance of morphology operations in microseconds on SD video using different number of SPE cores (on the IBM QS22 Blade) to characterize scalability.

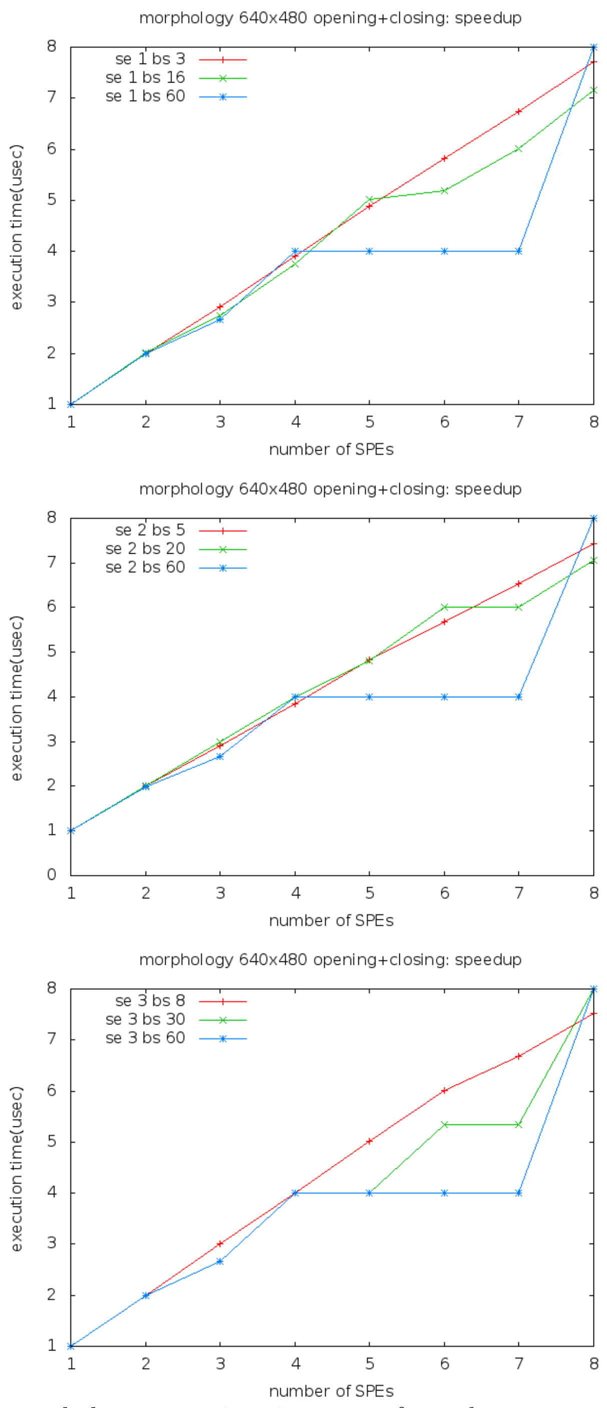


Figure 27: Performance of morphology operations in terms of speed-up compared to a one SPE core on SD video using different number of SPE cores (on the IBM QS22 Blade) to characterize scalability.

these four CCL algorithm applied to different sized images, with the maximum number of regions set to 250 and the total amount of foreground pixels set to 30% for all images. It was observed that *sequential_FW* gave better performance than *sequential_UnionFind* probably because branch instructions and recursions in *sequential_UF* code are not efficiently supported for execution. The average speedup for the parallel implementation without using the SIMD vectorization was about a factor of two. But when the SIMD instructions are used, the speedup is significantly higher improving to about 5 to 6 times faster. We measured how the parallel performance scales as the number of SPEs is varied. It can be observed that the execution time for the parallel implementation without SIMD instructions is greater than the serial execution time up to 3 SPEs due to the overhead of thread creation, data communication, merging the results, etc. which does not scale up linearly for a low number of SPE cores. Moreover the variation is not smooth as the total time is bounded by the maximum execution time amongst all the SPEs. By varying the number of SPEs, the data chunk partitioning area/boundary changes, which changes the number of regions or data chunks and consequently the number of foregrounds pixels (connected components) that needs to be processed by one SPE depending on the distribution pattern of components in the image. This also explains the unexpected increase in the execution time for 6 versus 5 SPEs. However, the optimized performance using SIMD instructions is always superior to that of the serial version and scales up consistently with increasing number of SPEs.

Thread creation and deletion was minimized by adding thread management code to reuse threads on SPEs during the code integration effort done by Black River Systems. This may have inadvertently introduced bugs that were later discovered by AFRL during testing (of the June 22, 2011 version of the code base). Testing was done using a Java program to create synthetic image frames with one to n simple color or grayscale objects of various sizes moving at various velocities (variable speed and direction). The bitmask produced by the flux tensor is correct so errors were shown to be restricted to the subsequent morphology, CCL or statistics modules. Some memory leak issues were addressed: within morphology.c, the function spu_morphology(), the heap was reset to zero whenever the function was called. This removed the extraneous Labels left in memory when the statistics function received a morphology frame. Within statistic.c, in computestats(), the heap is reinitialized to zero whenever it is called to remove the cumulative problem with statistics. But this caused the appearance of extraneous labels during further testing. In the function ccl( ), the variable *ymax* remains stuck at its original lower right position and does not appear to be updated correctly, whereas, *xmin*, *ymin*, and *xmax* appear to be updated as expected. Some sample debugging output from the integrated testing is shown below that exhibit these problems:

Listing 4: Row column indexing problem in morphology implementation

```
run_mode is 0
threshold is 100.000000
Connected to session management service
initialized blobextraction
Entering Connection Loop
Connected to session management service
Input channel started
Registered predicate
do_blob_extraction
Calling morphology
Calling statistics
PPU: spe 273177152 Total_ind 1
PPU: spe 273233776 Total_ind 1
PPU: spe 273238736 Total_ind 1
PPU: spe 273237728 Total_ind 1
PPU: spe 273256224 Total_ind 1
PPU: spe 273259488 Total_ind 1
PPU: spe 273177200 Total_ind 1
PPU: spe 273175584 Total_ind 1
Label=1; Area= 773; Centroid= [1900 1053]; BB= [1884 1039 1918 1066]
num_stats=1
```

```
Rate = 16.141002
do_blob_extraction
Calling morphology
Calling statistics
PPU: spe 273177152 Total_ind 1
PPU: spe 273233776 Total_ind 1
PPU: spe 273238736 Total_ind 1
PPU: spe 273237728 Total_ind 1
PPU: spe 273256224 Total_ind 1
PPU: spe 273259488 Total_ind 1
PPU: spe 273177200 Total_ind 1
PPU: spe 273175584 Total_ind 1
Label=1; Area= 1582; Centroid= [1900 1053]; BB= [1883 1039 1918 1066]
num_stats=1
Rate = 17.758704
do_blob_extraction
....
Calling morphology
Calling statistics
PPU: spe 273177152 Total_ind 1
PPU: spe 273233776 Total_ind 1
PPU: spe 273238736 Total_ind 1
PPU: spe 273237728 Total_ind 1
PPU: spe 273256224 Total_ind 1
PPU: spe 273259488 Total_ind 1
PPU: spe 273177200 Total_ind 1
PPU: spe 273175584 Total_ind 1
Label=1; Area= 52460; Centroid= [1849 1009]; BB= [1779 946 1918 1066]
num_stats=1
Rate = 21.458542
do_blob_extraction
Calling morphology
Calling statistics
...
num_stats=18
Rate = 19.884500
do_blob_extraction
Calling morphology
Calling statistics
PPU: spe 273177152 Total_ind 18
PPU: spe 273233776 Total_ind 18
PPU: spe 273238736 Total_ind 18
PPU: spe 273237728 Total_ind 18
PPU: spe 273256224 Total_ind 18
PPU: spe 273259488 Total_ind 18
PPU: spe 273177200 Total_ind 18
PPU: spe 273175584 Total_ind 18
Label=1; Area= 610708; Centroid= [1151 488]; BB= [0 541 1919 1066]
Label=2; Area= 33807; Centroid= [1145 589]; BB= [0 549 1919 966]
Label=3; Area= 19210; Centroid= [294 370]; BB= [0 557 1919 938]
Label=4; Area= 15839; Centroid= [403 270]; BB= [0 565 1919 938]
Label=5; Area= 10709; Centroid= [287 354]; BB= [0 573 1919 938]
Label=6; Area= 14765; Centroid= [332 333]; BB= [57 579 1919 938]
Label=7; Area= 12687; Centroid= [409 255]; BB= [0 581 1919 938]
```

```
Label=8; Area= 14801; Centroid= [403 270]; BB= [0 589 1919 938]
Label=9; Area= 11518; Centroid= [377 280]; BB= [0 597 1919 938]
Label=10; Area= 11761; Centroid= [361 301]; BB= [0 605 1919 938]
Label=11; Area= 10573; Centroid= [273 372]; BB= [0 613 1919 938]
Label=12; Area= 15045; Centroid= [338 333]; BB= [0 621 1919 938]
Label=13; Area= 11518; Centroid= [335 325]; BB= [0 629 1919 938]
Label=14; Area= 10287; Centroid= [347 311]; BB= [0 637 1919 938]
Label=15; Area= 8433; Centroid= [270 369]; BB= [0 645 1919 938]
Label=16; Area= 9528; Centroid= [314 341]; BB= [0 653 1919 938]
Label=17; Area= 7295; Centroid= [264 369]; BB= [0 661 1919 938]
Label=18; Area= 2643; Centroid= [324 236]; BB= [0 669 1919 938]
```

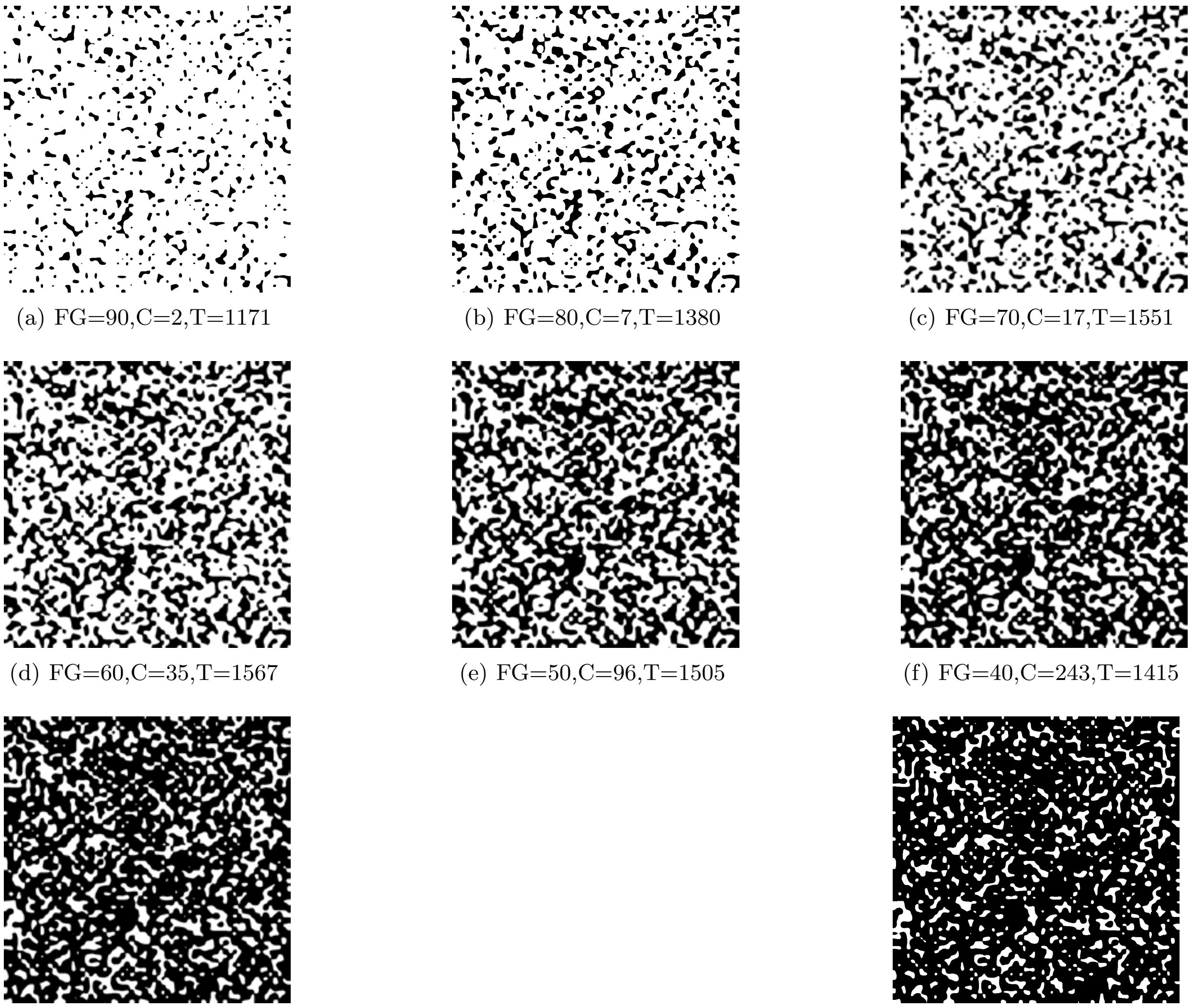
For evaluating the standalone implementation of the connected component algorithm, we generated test dataset of random images by introducing random gaussian noise and then dilating and resizing it to $1k \times 1k$ to increase the size of each components. Using the cumulative histogram of the resultant image, the threshold value was varied to obtain binary images with varying percentage of foreground pixels. Figure 28 shows some sample test images from the random noise blob dataset with varying percentage of foreground pixels, intermediate labels and components. We also generated another test dataset of diagonal lines (width greater than 2 pixels) which produces large number of intermediate labels for 4-connected component labeling and is useful for evaluating the scalability of algorithm as the number of intermediate labels grows. Test images of diagonal lines were generated using a sinusoidal image generator function using Eq. 25 with parameter values $\theta = \pi/4$ and $\omega = 10$; thresholding the resultant images with a threshold value between -1 to 1 produced binary images with a varying percentage of foreground pixels.

$$f(x,y) = \sin(2\pi\omega(x\cos(\theta) + y\sin(\theta))) \tag{25}$$

Figure 29 shows several diagonal line test images generated using Eq. 25.

Our CCL workload can be characterized in terms of following parameters: Percentage of foreground pixels (FG) in image, number of connected components (C), total number of intermediate labels (T), average labels per component (T/C) and average labels/SPE (A). Increase in FG, increases the time taken for scanning and establishing equivalences. Increase in C or T increases the operations for maintaining identities and resolving equivalences. The ratio T/C gives average number of labels per component, reflecting the complexity of shape and increases the time taken for resolving equivalences. Finally A gives a measure of average labels that each SPE has to handle in case of parallel execution. Figure 30 shows how the workload in the random blob test dataset varies in terms of these parameters for 8-connected analysis. C decreases with increase in FG pixels, except at the extreme zero. T is relatively lower at the ends and higher in between. A remains much less in comparison to T due to division among 6 SPEs with the standard deviation S of distribution of labels to SPEs as shown in Figure 30. For the test images with $FG \leq 50$, A remains small (less than 15) and are thus relatively less complex for resolving equivalences as compared to test images with $FG > 50$. It should be noted here in the context of Floyd-Warshall algorithm that when there are more number of labels that are equivalent, the operations in the innermost loop becomes more. For the diagonal line dataset the number of components is equal to 146 and the number of intermediate labels is equal to $74,523$ using 4-connected algorithm and the percentage of foreground pixels is varied by changing the thickness of the lines.

We compared the performance with different versions of the sequential and parallel code. We implemented sequential code using *Union-Find* algorithm (referred as *sequential_UF*) and *Floyd-Warshall* algorithm (referred as *sequential_FW*) for execution only on the PPE. The proposed parallel implementation for *Union-Find* algorithm is done without and with using SIMD instructions for labeling during first scan, referred to respectively as *parallel_UF* and *parallel_UF_SIMD*. The proposed parallel implementation for *Floyd-Warshall* algorithm is also done with and without using SIMD instructions for both labeling and resolution of equivalence class labels, which we refer to as *parallel_FW* and *parallel_FW_SIMD*, respectively. Figure 31 shows the performance results of these six different implementations on the random blob datasets by executing the sequential algorithms on the PPE and the parallel algorithms on the 6 SPE's of the PS3 with the PPE handling the control and merge tasks. We use a logarithmic scale to plot the execution



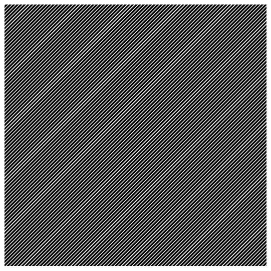
(a) FG=90,C=2,T=1171 (b) FG=80,C=7,T=1380 (c) FG=70,C=17,T=1551

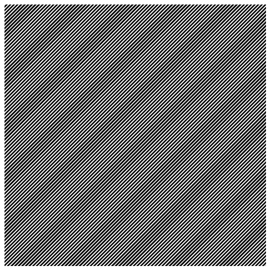
(d) FG=60,C=35,T=1567 (e) FG=50,C=96,T=1505 (f) FG=40,C=243,T=1415

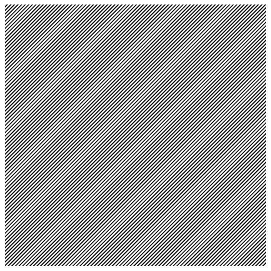
(g) FG=30,C=382,T=1174 (h) FG=20,C=483,T=910

Figure 28: Sample test images from random blob dataset, with varying percentage of foreground pixels (FG), number of connected components (C) and total intermediate labels (T), used for CCL testing and performance measurement.

times on the y-axis in order to capture the range of both sequential and parallel implementations on the same graph; so it should be noted that the differences become quite large as T increases. It was observed that the time taken for $sequential\_FW$ increases rapidly as T becomes large due to $O(n^3)$ complexity whereas the $sequential\_UF$ shows almost a linear increase and thus outperforms the $sequential\_FW$. The $sequential\_FW$ for images with $FG \geq 50$ percent increases more than for $FG < 50$ even though T decreases gradually for $FG \geq 50$. This can be understood by observing that the ratio T/C increases for $FG \geq 50$ explaining the additional operations in the innermost loop for resolving the redundant labels per component. The $parallel\_UF$ also shows almost a linear performance speedup of about 4.5 compared to its sequential implementation. The $parallel\_FW$ shows high variability in performance depending on the amount of work per SPEs; the average number of labels per SPE (A) is one-sixth of T and thus it speeds up the processing, especially for $FG > 50$ where the $sequential\_FW$ becomes very slow, reaching a maximum speedup of 30.7 at $FG = 90$. The $parallel\_UF\_SIMD$ improves the performance further by a factor of 1.4 with respect to $parallel\_UF$ due to vectorizing the scanning step and thus achieving average speedup of about 5.5 times


(a) FG=20,C=146,T=74,523


(b) FG=40,C=146,T=74,523


(c) FG=60,C=146,T=74,523

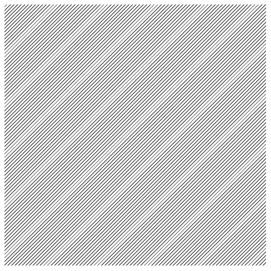

(d) FG=80,C=146,T=74,523

Figure 29: Sample test images from diagonal lines dataset with varying percentage of foreground pixels for CCL testing and performance measurement

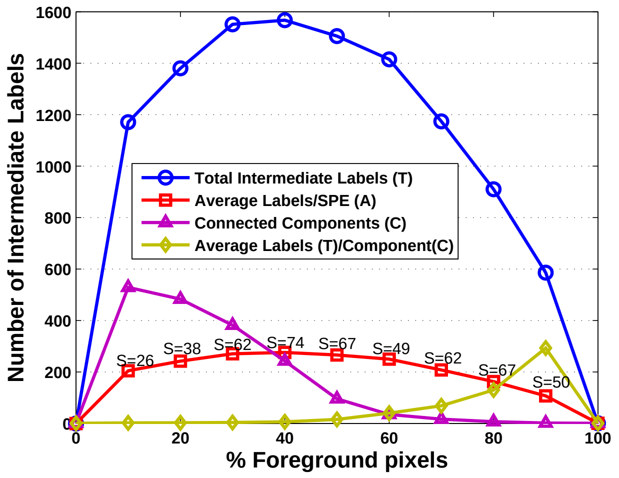


Figure 30: Variation of parameters characterizing our test dataset for CCL workload

with respect to *sequential_UF*. However, *parallel_FW_SIMD* improves the performance drastically with respect to *parallel_FW* and *sequential_FW* due to the use of SIMD instructions for vectorizing the label resolution step which involved the bulk of computation. The speedup achieved was variable with maximum of about 729 at $FG = 60$. Overall, *parallel_UF_SIMD* achieved the best performance for most of the time with moderate speedup but *parallel_FW_SIMD* achieved the best speedup with performance very close to *parallel_UF_SIMD* and even better when $A < 150$. On experimenting with the diagonal lines dataset, we found that the *Floyd-Warshall* algorithm could not scale up to handle such a large number of intermediate labels $(74, 523)$ both in terms of memory requirements especially on the SPU local store and processing time.

On the other hand, the *Union-Find* algorithm is able to scale well to handle large intermediate labels by flattening the tree every time so that almost linear performance is achieved as in the case of the random noise blob dataset. Figure 32 shows the performance result of the sequential and parallel versions of *Union-Find* algorithm on the diagonal line dataset. In this case, the speedup achieved without using SIMD is on average 4.5 and with using SIMD is on average 6.5.

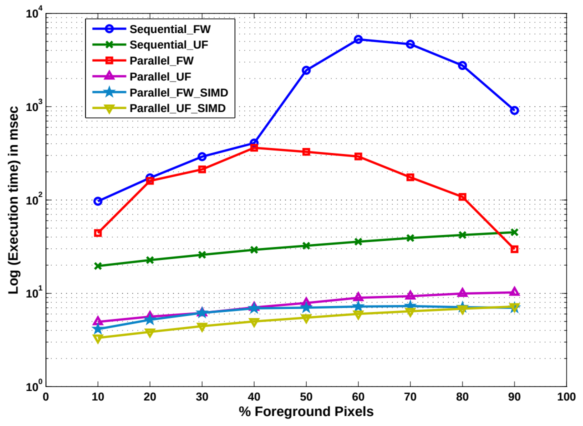


Figure 31: Execution time per frame for six different implementations of CCL on our random dataset. For parallel version, execution time includes DMA transfers of inputs to and outputs from SPE.

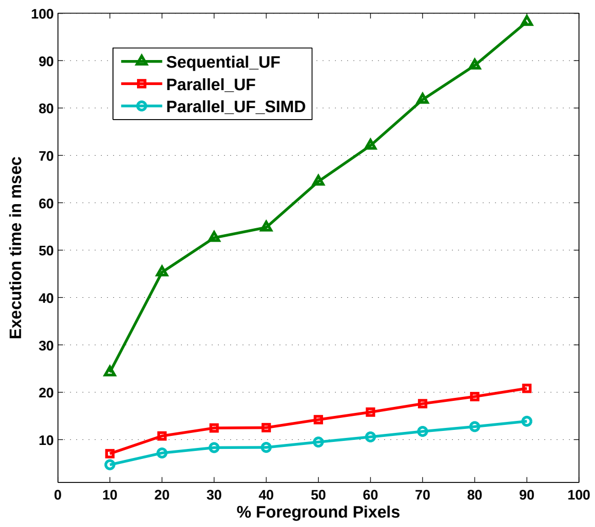


Figure 32: Execution time per frame for different implementations of *Union-Find* CCL algorithm on diagonal line dataset. For parallel version, execution time includes DMA transfers of inputs to and outputs from SPE.

Figure 33 shows the performance of the six different sequential and parallel CCL algorithms using five different input image sizes. The binary image was selected from the random dataset with foreground pixels occupying 30% of the image and it was rescaled to different sizes. The execution time is plotted on logscale because the time taken by sequential FW algorithm increases exponentially due to large number of intermediate labels. Figure 34 shows the speedup of the parallel CCL algorithms using 6 SPEs, compared to the corresponding sequential versions (UF and FW) running on the PPE which converts the timing information presented in Figure 33 to speedup ratios (plotted on logscale). Similar speedup pattern was observed amongst different implementations but it can be observed that the speedup obtained for 1280×780 and HD

Table 5: Comparison of execution times for relabeling and statistic computation done on PPE (sequential) versus SPE (parallel).

Image Size	**Sequential** (time in msec)	**Parallel** (time in msec)	**Speedup**
512 x 512	3.8	1.3	2.9
720 x 480	5.4	1.6	3.5
1280 x 720	15.12	3.5	4.3
1024 x 1024	16.53	4.1	4.0
1920 x 1080	32.6	6.5	5.0

image size is larger than others due to larger width to height aspect ratio. Increase in the width achieves more parallelization than increase in the height of the image because of row wise partitioning and merging. Figure 35 shows how the parallel performance scales as the number of SPEs is varied using the same test image. It can be observed that the execution times for the parallel implementation of the FW and UF algorithms without the use of SIMD instructions, using up to five and two SPEs respectively, are greater than the serial execution time due to the overhead of thread creation, data communication, etc. Moreover the variation is not smooth as the total time is bounded by the maximum of the execution time of all the SPE. By varying the number of SPE's, the data partitioning area/boundary changes, which can cause increase or decrease in the number of regions and foreground pixels to be processed by any SPE depending on the distribution pattern in the image. However, the fully optimized implementation with the use of SIMD instructions is always superior to the serial version and it scales up consistently with increasing number of SPEs. Similar performance was measured on the QS20 and QS22 servers when using only a single processor with six of the eight SPEs active. This is again because the algorithm involves only comparison operations and does not benefit from the additional memory or improved double precision floating point performance of the QS20 and QS22 servers.

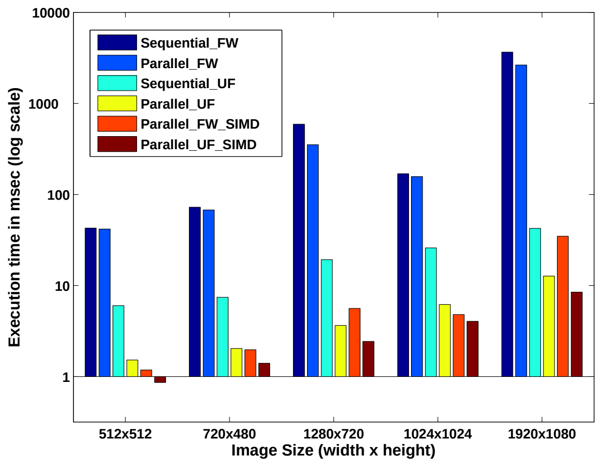


Figure 33: Execution time per frame for six different implementations of CCL. For parallel version, execution time includes DMA transfers of inputs to and outputs from SPE.

Table 5 compares the time taken for statistics computation and relabeling done sequentially on PPE against parallel implementation on SPEs for the same random noise test image with FG=30% and 382 number of components. The speedup increases for larger size images (up to a factor of 5 for HD sizes) as the overhead for parallelization decreases compared to the amount of computation. It can be observed that although this part of computation involves simple arithmetic operations and memory access but the execution time becomes significantly large for larger size images and thus doing it in parallel makes it more

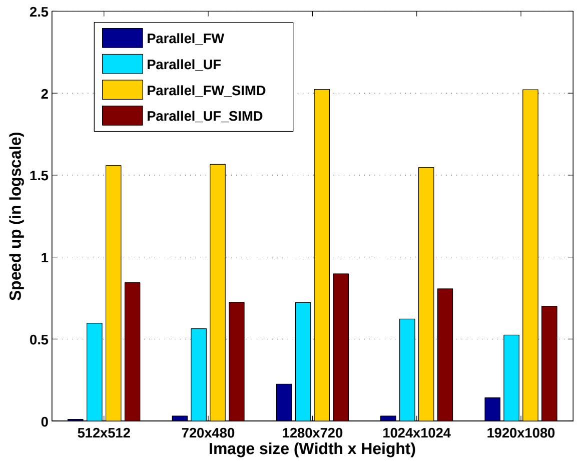


Figure 34: Speedup of parallel implementations of CCL with respect to their sequential counterpart. For parallel version, execution time includes DMA transfers of inputs to and outputs from SPE.

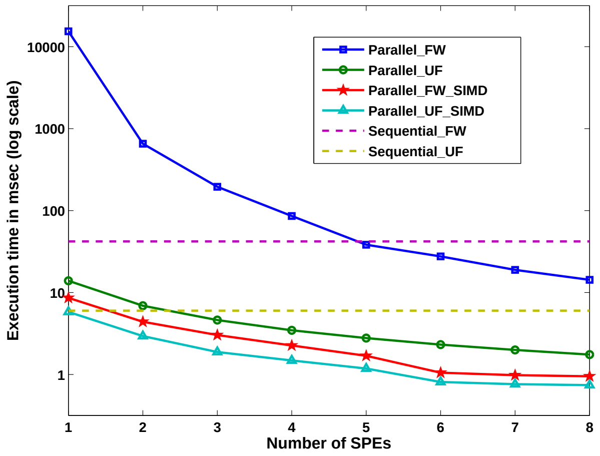


Figure 35: Variation in parallel CCL performance across different number of SPEs on Cell. Execution time includes DMA transfers of inputs to and outputs from SPE

efficient.

For testing the performance of the whole system with the interconnection of the different processing modules on the Cell processor, we had to consider some important issues. The limited local memory of SPEs could not accommodate the code and the data for all the modules. To execute same task on all the SPE's at a time and completing the entire chain of processing in the successive sequence of tasks required re-initializing the SPE's again and again with different code which produced lot of overhead due to code loading, thread initialization, synchronization etc. Therefore, we divided the processing into two groups for execution on two PS3 Cell processors so as to achieve real-time processing on the HD video. We dedicate one Cell processor for flux tensor computation only as it is most expensive and we combine the blob extraction modules (morphology + CCL + statistics) together for execution on the second Cell processor. Since the output from the flux tensor is a compressed foreground mask with one pixel represented by one bit, the data communication between the Cell processor did not raise a serious concern. We observed that for HD size video frames, the flux tensor on the first Cell processor produced output at the rate of about 41 frames/sec (previous version) and the blob extraction modules on the second Cell processor produced the output at the rate of about 38 frames/sec. We exclude the IO time for fetching the image frames for the flux tensor and

Table 6: Performance summary of major algorithms at different stages of moving object detection workload prior to integration of the code base. The Cell performance is measured using only 6 SPEs on the PS3.

Algorithm	**Flux Tensor Motion**		**Morphology (1 Opening + 1 Closing)**		**CCL**	
Specification	Deriv. filters-(7,7,5), Avg. filters- (7,7,5)		Strel. 7 x 7 Ellipse		FG 30%,Components - 382	
Image Size	640 × 480	1920 × 1080	640 × 480	1920 × 1080	640 × 480	1920 × 1080
Sequential Code	12.37 fps (80.8 msec)	1.6 fps (0.625 sec)	7.36 fps (0.135 sec)	1.04 fps (0.961 sec)	134.7 fps (7.42 msec)	23.5 fps (42.42 msec)
Parallel on Cell	243.1 fps (4.12 msec)	54 fps (18.5 msec)	4464.2 fps (0.224 msec)	688.7 fps (1.45 msec)	709.2 fps (1.41 msec)	118.2 fps (8.46 msec)
Operation count	177.7 Mflop/frame	1.2 Gflop/frame	123.8 Mega comparisons/frame	836	36.5 Mega comparisons/frame	219
Speedup on Cell	19.65	33.2	616.4	660.6	5.3	5.0

the communication of flux output to the blob extraction module in our performance measurement.

We provided a description of the parallel algorithms for flux tensor motion estimation, binary image morphology and connected component labeling optimized for execution on multicore Cell processor. A significant amount of implementation time was devoted to memory management and data alignment, data movement (DMA transfers), vectorization and shuffling issues. Since the SPU has only 256KB of memory and limited programmer development tools, this required manual code partitioning and coordinating data movement into and out of the SPE registers. Flux tensor calculations involving separable 3D convolutions were done using fused vector multiply adds (FMA) to approach the peak SPU performance. Data partitioning and loop unrolling required careful tuning of implementation parameters like work unit data chunking, number of stages to unroll loops that is constrained by the amount of available SPE local store memory, 6 cycle latency in the FMA unit and other factors. For using SIMD instructions, additional shuffle operations are required to transfer data into the SPE registers that may not be aligned to 128-bit vector cache lines. For HD image size and filter configuration of $n_{D_s} = 7, n_{D_t} = 5, n_{A_s} = 7, n_{A_t} = 5$ a speedup of about 33 times was observed on the Cell BE.

The binary morphology operations were optimized for execution on the Cell SPE by representing each pixel by a single bit and utilizing bitwise comparison AND/OR SIMD operations to process 128 pixels simultaneously. Novel bit level data access and alignment techniques for Cell BE were proposed in this context. Our implementation using 6 SPE's achieves a speedup of up to 660 times for processing erosion/dilation operation using 7×7 pixels size kernels on HD size image. The parallelization approach for CCL algorithm proposed a suitable data partitioning and merging algorithm for execution on PPE and multiple SPEs. We implemented parallel versions of the Union-Find and Floyd-Warshall algorithms for labeling and equivalence label resolution on the SPE side. Three different variations for both algorithms Viz. sequential, parallelized and vectorized using SIMD operation were implemented and tested on a dataset with varying workload characteristics to test the performance. The CCL implementation using the Union-Find algorithm achieved the best performance for most cases due to its linear time behavior with the parallel version showing a moderate speedup of about 5 compared to the sequential version. The Floyd-Warshall algorithm performed well only when the intermediate labels remain low (< 150) and degrades rapidly with increase in number of labels. However the parallel implementation using SIMD instructions for resolving equivalence achieves maximum speedup, especially when the total intermediate labels becomes large and closely follows the best performance of the parallel implementation using Union-find. An average speed up of about 5 times was achieved for labeling the random noise image with 30% foreground pixels and 382 components. Statistic computation and relabeling could also be speedup to the factor of 5 by parallel computation on SPEs for HD size image. Table 6 shows a workload summary of major algorithms implemented as separate modules (prior to integration) at different stages of moving object detection and the performance gain achieved by the parallel implementation on Cell for HD size images. The parallel performance observed on PS3, QS20 Cell Blade and QS22 PowerXCell Blade was observed to be the same with using only 6 SPEs (as in PS3) for our algorithms. The system processes HD sized video frames in real time using two PS3 Cell processors.

5. CONCLUSIONS

Parallel video processing algorithms in the context of N-CET sensor pod requirements were implemented for object tracking in high definition video streams at a data capture bandwidth of 1.5Gb/s. Parallelization of algorithms was primarily focused on the IBM Cell/B.E. class of multicore architectures due to its high performance to power efficiency ratio (PPR), low cost and representativeness of other similar architectures for high performance computing. The specific set of end-to-end video processing chain of algorithms was implemented in a modular fashion for rapid prototyping followed by inclusion in the AFRL N-CET specific software environment. Multicore algorithms implemented to exploit fine-grained parallelism included flux tensor for motion estimation to identify potential moving objects, frame averaging and thresholding, binary image morphology for pre- and post-processing to reduce noise, connected component labeling to group image pixels into moving blobs, and object statistics after blob extraction to characterize blobs for real time tracking. We assisted with integrating the Cell/B.E. processor modular codes with the Phoenix pub-sub high performance database, reduced thread overhead, improved video processing module interoperability, enhanced video streaming performance, evaluated power requirement tradeoffs between different algorithms, assisted with tighter integration of the individual modules across multiple Cell/B.E. processors and gathered benchmarking data to analyze performance bottlenecks in communication pathways and within computational modules. The video processing modules were ported to the IBM QS20/QS22 Blade architectures with 16 SPEs for improved numerical computation performance, especially for the flux tensor computation.

A significant amount of implementation time was devoted to memory management and data alignment, data movement (DMA transfers), vectorization and shuffling issues. Since the SPU has only 256KB of memory and limited programmer development tools this required manual code partitioning and coordinating data movement into and out of the SPE registers. Flux tensor calculations involving separable 3D convolutions were done using fused vector multiply adds and filter specific optimizations to approach the peak SPU performance. The flux tensor SPE implementation previously partitioned the data into vertical data blocks or strips for each SPE. The vertical data chunking does not match with the horizontal data block-based implementations for the other modules (morphology, CCL, statistics). So the flux tensor implementation was modified to use horizontal data blocks in line with the rest of the interprocess data flow patterns. The flux tensor component operators were further reordered to perform the temporal derivatives first which significantly reduced the number of computations, reduced memory requirements and sped up the algorithm by a factor of two for the smallest filter sizes to about six for the larger filter sizes.

Data partitioning and loop unrolling required careful tuning of implementation parameters like work unit data chunking, number of stages to unroll loops that is constrained by the amount of available SPE local store memory, 6 cycle latency in the FMA unit and other factors. For using SIMD instructions, additional shuffle operations are required to transfer data into the SPE registers that may not be aligned to 128-bit vector cache lines. For HD image size and filter configuration of $n_{D_s} = 7, n_{D_t} = 5, n_{A_s} = 7, n_{A_t} = 5$ a speedup of about 33 times was observed on the Cell BE.

The binary morphology operations were optimized for execution on the Cell SPE by representing each pixel by a single bit and utilizing bitwise comparison AND/OR SIMD operations to process 128 pixels simultaneously. Novel bit level data access and alignment techniques for Cell BE were proposed in this context. Our implementation using 6 SPE's achieved a speedup of up to 660 times for processing erosion/dilation operation using 7×7 pixels size kernels on HD size image. The parallelization approach for CCL algorithm proposed a suitable data partitioning and merging algorithm for execution on PPE and multiple SPEs. We implemented parallel versions of the Union-Find and Floyd-Warshall algorithms for labeling and equivalence label resolution on the SPE side. Three different variations for both algorithms Viz. sequential, parallelized and vectorized using SIMD operation were implemented and tested on a dataset with varying workload characteristics to test the performance. Implementation using Union-Find algorithm achieved the best performance for most cases due to its linear time behavior and the parallel implementation achieved a moderate speedup of about 5. The Floyd-Warshall algorithm performed well only when the intermediate labels remain low (< 150) and degrades rapidly with increase in number of labels. However, the proposed parallel implementation using SIMD instructions for resolving equivalence achieves maximum speedup, especially when the total intermediate labels becomes large; and closely follows the best performance of the parallel implementation using Union-Find. An average speed up of about 5 times was achieved for labeling the random noise image with 30% foreground pixels and 382 components. Statistic computation and relabeling could also be speedup to the factor of 5 by parallel computation on SPEs for HD size image. Table 6 presents a summary of characteristics of major algorithms implemented in this report at different stages of moving object detection workload and the performance gain achieved by the parallel implementation on Cell for HD size image. The parallel performance observed on PS3, QS20 Cell Blade and QS22 PowerXCell Blade was observed to be the same with using only 6 SPEs (as in PS3) for our algorithms. The whole system can process HD size video frames in real time using two PS3 Cell processors. The proposed implementation can be used to support N-CET class large scale video object detection and tracking applications that require scalable processing for sensor networks and distributed dissemination of data and processing results up to HD resolution.

6. REFERENCES

[1] S. Albers. Energy efficient algorithms. *Communications of the ACM*, 53(5), May 2010.

[2] S. Albers, F. Muller, and S. Schmelzer. Speed scaling on parallel processors. In *Proc. 19^{t}h ACM Symposium on Parallelism in Algorithms and Architectures*, pages 289–298, 2007.

[3] S. Ali and M. Shah. COCOA - Tracking in aerial imagery. In Daniel J. Henry, editor, *SPIE Airborne Intelligence, Surveillance, Reconnaissance (ISR) Systems and Applications III*, number 6209 in Proceedings of the SPIE, page Online, 2006.

[4] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and P.A. Wacrenier. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23:187–198, 2011.

[5] D. A. Bader and V. Agarwal. FFTC: Fastest Fourier transform for the IBM Cell Broadband Engine. *Lecture Notes in Computer Science (HiPC)*, 4873:172–184, 2007.

[6] Pieter Bellens, Josep M Perez, Rosa M Badia, and Jesus Labarta. CellSs : A Programming Model for the Cell BE Architecture. In *Proc. ACM/IEEE Conference Supercomputing*, 2006.

[7] B. Bouzas, R. Cooper, J. Greene, M. Pepe, and M-J. Prelle. Multicore framework: An API for programming heterogeneous multicore processors. Technical report, Mercury Computer Systems, Inc., 2006.

[8] F. Bunyak, K. Palaniappan, S. K. Nath, and G. Seetharaman. Flux tensor constrained geodesic active contours with sensor fusion for persistent object tracking. *J. Multimedia*, 2(4):20–33, August 2007.

[9] F. Bunyak, K. Palaniappan, S.K. Nath, and G. Seetharaman. Geodesic active contour based fusion of visible and infrared video for persistent object tracking. *IEEE Workshop Applications of Computer Vision (WACV 2007)*, page Online, 2007.

[10] A. Buttari, P. Luszczek, J. Kurzak, J. Dongarra, and G. Bosilca. A rough guide to scientific computing on the Playstation 3. Technical Report UT-CS-07-595, Innovative Computing Laboratory, University of Tennessee Knoxville, 2007.

[11] A. L. Chan. A description on the second dataset of the U.S. Army Research Laboratory Force Protection Surveillance System. Technical Report ARL-MR-0670, Army Research Laboratory, Adelphi, MD, 2007.

[12] T. P. Chen, D. Budnikov, C. J. Hughes, and Yen-Kuang Chen. Computer vision on multi-core processors: Articulated body tracking. *IEEE International Conference on Multimedia and Expo*, pages 1862–1865, 2007.

[13] R. T. Collins, A. J. Lipton, T. Kanade, H. Fujiyoshi, D. Duggins, Y.Tsin, D. Tolliver, N. Enomoto, O. Hasegawa, P. Burt, and L.Wixson. VSAM: A system for video surveillance and monitoring. PA, Technical Report CMU-RI-TR-00-12, Carnegie Mellon Univ., Pittsburgh, 2000.

[14] R.T. Collins, A.J. Lipton, H. Fujiyoshi, and T. Kanade. Algorithms for cooperative multisensor surveillance. *Proc. of the IEEE*, 89:1456–1477, October 2001.

[15] R.T. Collins, Y. Liu, and M. Leordeanu. Online selection of discriminative tracking features. *IEEE Trans. Pattern Anal. Mach. Intel.*, 27(10):1631–1643, Oct. 2005.

[16] K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. R. Horn, L. Leem, J-Y. Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the memory hierarchy, p. online. *Proc. ACM/IEEE Conference Supercomputing*, 2006.

[17] S. Grauer-Gray, C. Kambhamettu, and K. Palaniappan. GPU implementation of belief propagation using CUDA for cloud tracking and reconstruction. In *5th IAPR Workshop on Pattern Recognition in Remote Sensing (ICPR)*, pages 1–4, 2008.

[18] M. Gschwind. The Cell Broadband Engine: Exploiting multiple levels of parallelism in a chip multiprocessor. *International Journal of Parallel Programming*, 35(3):233–262, 2007.

[19] A. Hafiane, K. Palaniappan, and G. Seetharaman. UAV-video registration using block-based features. In *IEEE Int. Geoscience and Remote Sensing Symposium*, volume II, pages 1104–1107, 2008.

[20] M. Hidemasa, D. Munehiro, N. Hiroki, and M. Yumi. Multilevel parallelization on the Cell/B.E. for a motion JPEG 2000 encoding server. *Proc. 15th International Conference Multimedia*, pages 942–951, 2007.

[21] B.P. Horn and B.G. Schunck. Determining optical flow. *Artificial Intell.*, 17(1-3):185–203, Aug. 1981.

[22] K. Jefferson and C. Lee. Computer vision workload analysis - case study of video surveillance systems. *Intel Technology Journal*, 09(02), 2005.

[23] P. Kumar, K. Palaniappan, A. Mittal, and G. Seetharaman. Parallel blob extraction using the multi-core Cell processor. *Lecture Notes in Computer Science (ACIVS)*, 5807:320–332, 2009.

[24] Jakub Kurzak, Alfredo Buttari, Piotr Luszczek, and Jack Dongarra. The PlayStation 3 for high-performance scientific computing. *IEEE Computing in Science and Engineering*, pages 84–87, May/June 2008.

[25] L. Liu, S. Kesavarapu, J. Connell, A. Jagmohan, A. Leem, L. Paulovicks, B. Sheinin, V. L. Tang, and H. Yeo. Video analysis and compression on the STI Cell broadband engine processor. *IEEE International Conference on Multimedia and Expo*, 2006.

[26] Y. Liu and K. N. Ngan. Fast multiresolution motion estimation algorithms for wavelet-based scalable video coding. *ACM Image Communication*, 22(5):448–465, 2007.

[27] M. Manohar and H.K. Ramapriyan. Connected component labeling of binary images on a mesh connected massively parallel processor. *Computer Vision, Graphics, and Image Processing*, 45(2):133–149, 1989.

[28] M. D. McCool. Data-parallel programming on Cell BE and the GPU using the rapidmind development platform, p. online. *In GSPx Multicore Applications Conference*, 2006.

[29] S. Mehta, A. Misra, A. Singhal, P. Kumar, A. Mittal, and K. Palaniappan. Parallel implementation of video surveillance algorithms on GPU architectures using CUDA. In *17th IEEE Int. Conf. Advanced Computing and Communications (ADCOM)*, 2009.

[30] J. M. Metzler, M. H. Linderman, and L. M. Seversky. N-CET: Network-centric exploitation and tracking. In *IEEE Military Communications Conf. (MILCOM)*, pages 1–7, 2009.

[31] S. Momcilovic and L. Sousa. A parallel algorithm for advanced video motion estimation on multicore architectures. *International Conference Complex, Intelligent and Software Intensive Systems*, pages 831–836, 2008.

[32] H.H. Nagel and A. Gehrke. Spatiotemporally adaptive estimation and segmentation of OF-Fields. In *LNCS-1407: Proc. 5th ECCV*, volume 2, pages 86–102, Freiburg, Germany, June 1998. Springer-Verlag.

[33] K. Palaniappan, I. Ersoy, and S. K. Nath. Moving object segmentation using the flux tensor for biological video microscopy. *Lecture Notes in Computer Science (PCM)*, 4810:483–493, 2007.

[34] K. Palaniappan, M. Faisal, C. Kambhamettu, and A. F. Hasler. Implementation of an automatic semi-fluid motion analysis algorithm on a massively parallel computer. *10th IEEE Int. Parallel Processing Symp.*, pages 864–872, 1996.

[35] K. Palaniappan, H. S. Jiang, and T. I. Baskin. Non-rigid motion estimation using the robust tensor method. In *IEEE CVPR Workshop on Articulated and Nonrigid Motion*, volume 1, pages 25–33, Washington DC, USA, June 27–July 2 2004.

[36] I. Pavlidis, V.Morellas, P. Tsiamyrtzis, and S. Harp. Urban surveillance systems: From the laboratory to the commercial world. *Proceedings of IEEE*, 89(10):1478–1497, 2001.

[37] J. M. Perez, P. Bellens, R. M. Badia, and J. Labarta. CellSs: making it easier to program the Cell broadband engine processor. *IBM Journal of Research and Development*, 51(5):593–603, 2007.

[38] P. Bellens J. M. Perez, , R. M. Badia, and J. Labarta. CellSs: A programming model for the Cell BE architecture. In *Proc. ACM/IEEE Conference Supercomputing*, 2006.

[39] A.C. Sankaranarayanan, A. Veeraraghavan, and R. Chellappa. Object detection, tracking and recognition for multiple smart cameras. *Proceedings of the IEEE*, 96(10):1606–1624, 2008.

[40] G. Seetharaman, G. Gasperas, and K. Palaniappan. A piecewise affine model for image registration in 3-D motion analysis. In *IEEE Int. Conf. Image Processing*, pages 561–564, 2000.

[41] H. Sugano and R. Miyamoto. Parallel implementation of morphological processing on Cell/BE with OpenCV interface. *3rd International Symposium Communications, Control and Signal Processing*, pages 578–583, 2008.

[42] Hai-Yun Wang and Kai-Kuang Ma. Spatio-temporal video object segmentation via scale-adaptive 3D structure tensor. *EURASIP J. Appl. Signal Process.*, 2004(6):798–813, 2004.

[43] C. Weele, H. Jiang, and et al. A new algorithm for computational image analysis of deformable motion at high spatial and temporal resolution applied to root growth. *Plant Physiology*, 132(3):1138–1148, July 2003.

[44] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. A. Yelick. Scientific computing kernels on the Cell processor. *Int. J. Parallel Programming*, 35:263–298, 2007.

[45] P. R. Woodward, J. Jayaraj, P-H. Lin, and P-C. Yew. Moving scientific codes to multicore microprocessor CPUs. *IEEE Computing in Science and Engineering*, 10(6):16–25, 2008.

[46] J. Yu and H. Wei. Video processing and retrieval on cell processor architecture. *Lecture Notes in Computer Science*, 4740:255–262, 2007.

[47] Z. Yue, D. Guarino, and R. Chellappa. Moving object verification in airborne video sequences. *IEEE Trans. Circuits and Systems for Video Technology*, 19(1):77–89, Jan. 2009.

[48] J. Zhang, J. Gao, and W. Liu. Image sequence segmentation using 3-D structure tensor and curve evolution. *IEEE Trans. Circuits and Systems for Video Technology*, 11(5):629–641, May 2001.

[49] X. Zhuang, Y. Huang, K. Palaniappan, and Y. Zhao. Gaussian mixture density modeling, decomposition and applications. *IEEE Trans. Image Processing*, 5(9):1293–1302, Sept. 1996.

[50] X. Zhuang, K. Palaniappan, and R. M. Haralick. Highly robust statistical methods based on minimum-error bayesian classification. In C. W. Chen and Ya-Qin Zhang, editors, *Visual Information Representation, Communication and Image Processing*, Optical Engineering, pages 415–430. Marcel-Dekker, 1999.

7. LIST OF ACRONYMNS

AFRL - Air Force Research Laboratory

Cell/B.E. - IBM Cell Broadband Engine multicore processor

CCL - connected component labeling

CPU - central processing unit

CUDA - Compute Unified Device Architecture (nVidia)

DMA - direct memory access

FLOPS - floating point operations per second

FMA - fused multiply add

FPSS - Force Protection Surveillance System video collection

GFLOPS - gigaflops (billion floating point operations per second)

GPU - graphics processing unit

HD - high definition

JBI - Joint Battlespace Infosphere database

JCAN - Joint Capability for Airborne Networking

JPEG - Joint Photographic Experts Group

KB - kilobyte (1024 bytes)

MB - megabyte

MBB - minimum bounding box

MFC - memory-flow controller

MFLOPS - megaflops (million floating point operations per second)

MHT - multiple hypothesis tracking

N-CET - Net-Centric Exploitation and Tracking

NP-hard - non-deterministic polynomial-time hard

Phoenix - follow-on to JBI

PS-3 - SONY Playstation 3

PPE - Power Processing Element

SAD - Sum of Absolute Differences

SB - subblock or macroblock

SD - standard definition

SIMD - single instruction

SPE - Synergistic Processing Element

SIMD - single instruction multiple data

SPMD - single program multiple data

WW - work width